Venue   *MEDLEY LANGUAGE REFERENCE*

MEDLEY REFERENCE MANUAL

VOLUME I: LANGUAGE

April, 1993

The information in this document is subject to change without notice and should not be construed as a commitment by Venue. While every effort has been made to ensure the accuracy of this document, Venue assumes no responsibility for any errors that may appear.

Text was written and produced with Venue text formatting tools; PostScript printers were used to produce masters. The typeface is Palatino.

TABLE of CONTENTS

Volume 1 - Lanuage Reference

Volume 2 - Environment Reference

22. Processes 23-1

Volume 3 - I/O Reference

23. Streams and Files 24-1

[This page intentionally left blank]

1. INTRODUCTION

Medley is a *programming system* that consists of a programming *language*, a large number of predefined programs (or *functions*) that you can use directly or as subroutines, and an *environment* that supports you with a variety of specialized programming tools. The language and predefined functions of Lisp are rich, but similar to those of other modern programming languages. The Medley programming environment, on the other hand, is very distinctive. Its main feature is an integrated set of programming tools that know enough about Interlisp and Common Lisp to act as semi-autonomous, intelligent "assistants" to you. This environment provides a completely self-contained world for creating, debugging and maintaining Lisp programs.

This manual describes all three parts of Medley. There are discussions of the language, about the pieces of the system that can be incorporated into your programs, and about the environment. The line between your code and the environment is thin and changing. Most users extend the environment with some special features of their own. Because Medley is so easily extended, the system has grown over time to incorporate many different ideas about effective and useful ways to program. This gradual accumulation over many years has resulted in a rich and diverse system. It is also the reason this manual is so large.

The rest of this manual describes the individual pieces of Medley; this chapter describes system as a whole—including the otherwise-unstated philosophies that tie it all together. It will give you a global view of Medley.

Lisp as a Programming Language

This manual is not an introduction to programming in Lisp. This section highlights a few key points about lisp that will make the rest of the manual clear.

In Lisp, large programs (or functions) are built up by composing the results of smaller ones. Although Medley, like most modern Lisps, lets you program in almost any style you can imagine, the natural style of Lisp is functional and recursive—each function computes its result by calling lower-level "building-block" functions, then passing that result back to its caller (rather than by producing "side-effects" on external data structures, for example).

Lisp is also a list-manipulation language. Like other languages, Lisp can process characters and numbers. But you get more power if you program at a higher level. The primitive data objects of Lisp are "atoms" (symbols or identifiers) and "lists" (sequences of atoms or lists), which you use to represent information and relationships. Each Lisp dialect has a set of operations that act on atoms and lists, and these operations comprise the core of the language.

Invisible in the programs, but essential to the Lisp style of programming, is an automatic memory management system (an "allocator" and a "garbage collector"). New storage is allocated automatically whenever a you create a new data object. And that storage is automatically reclaimed for reuse when no other object refers to it. Automated memory management is essential for rapid,

large-scale program development because it frees you from the task of maintaining the details of memory administration, which change constantly during rapid program evolution.

A key property of Lisp is that Lisp function definitions are just pieces of Lisp list data. Each subfunction "call" (or *function application*) is written as a list with the function first, followed by its arguments. Thus, `(PLUS 1 2)` represents the expression `1+2`. A function's definition, then, is just a list of such function applications, to be evaluated in order. This representation of program as data lets you use the same operations on programs that you use on data—making it very easy to write Lisp programs that look at and change *other Lisp programs*. This, in turn, makes it easy to develop programming tools and translators, which was essential to the development of the Medley environment.

The most important benefit of this is that you can extend the Lisp programming language itself. Do you miss some favorite programming idiom? Just define a function that translates the desired expression into simpler Lisp. Now your idiom is *part of the language*. Medley has extensive facilities for making this type of language extension. Using this ability to extend itself, Interlisp has incorporated many of the constructs that have been developed in other modern programming languages (e.g. if-then-else, do loops, etc.).

Medley as an Interactive Environment

Medley programs should not be thought of as simple files of source code. All Medley programming takes place within the Medley environment, which is a completely self-sufficient environment for developing and using Medley programs. Beyond the obvious programming facilities (e.g., program editors, compilers, debuggers, etc.), the envionrment also contains a variety of tools that "keep track" of what happens. For example, the Medley File Manager notices when programs or data have been changed, so the system will know what needs to be saved at the end of a session. The "residential" style, where you stay inside the environment throughout the development, is essential for these tools to operate. Furthermore, this same environment is available to support the final production version, some parts providing run time support and other parts being ignored until the need arises for further debugging or development.

For terminal interaction, Medley provides a top level "Read-Eval-Print" executive, which reads whatever you type in, evaluates it, and prints the result. (This interaction is also recorded, so you can ask to do an action again, or even to undo the effects of a previous action.) Although Executives understand some specialized commands, most of the interaction will consist of simple Lisp expressions. So rather than special commands for operations like manipulating your files, you just type the same expressions that you would use to accomplish them in a Lisp program. This creates a very rich, simple, and uniform set of interactive commands, since any Lisp expression can be typed at an executive and evaluated immediately.

In normal use, you write a program (or rather, "define a function") by typing in an expression that invokes the "function defining" function `(DEFINEQ)`, giving it the name of the function being defined and its new definition. The newly-defined function can be executed immediately, simply by using it in a Lisp expression.

In addition to these basic programming tools, Medley also provides a wide variety of programming support mechanisms:

List structure editor Since Lisp programs are represented as list structure, Medley provides an editor which allows one to change the list structure of a function's definition directly. See Chapter 16.

Pretty-printer The pretty printer is a function that prints Lisp function definitions so that their syntactic structure is displayed by the indentation and fonts used. See page Chapter 26.

Debugger When errors occur, the debugger is called, allowing you to examine and modify the context at the point of the error. Often, this lets you continue execution without starting from the beginning. Within a break, the full power of Interlisp is available to you. Thus, the broken function can be edited, data structures can be inspected and changed, other computations carried out, and so on. All of this occurs in the context of the suspended computation, which remains available to be resumed. See Chapter 14.

DWIM The "Do What I Mean" package automatically fixes misspellings and errors in typing. See Chapter 20.

Programmer's Assistant Medley keeps track of your actions during a session and allows each one to be replayed, undone, or altered. See Chapter 13.

Masterscope Masterscope is a program analysis and management tool which can analyze users' functions and build (and automatically maintain) a data base of the results. This allows you to ask questions like "`WHO CALLS ARCTAN`" or "`WHO USES COEF1 FREELY`" or to request systematic changes like "`EDIT WHERE ANY` [function] `FETCHES ANY FIELD OF` [the data structure] `FOO`". See Chapter 19.

Record/Datatype Package Medley allows you to define new data structures. This enables one to separate the issues of data access from the details of how the data is actually stored. See Chapter 8.

File Manager Source code files in Medley are managed by the system, removing the problem of ensuring timely file updates from the user. The file manager can be modified and extended to accomodate new types of data. See Chapter 17.

Performance Analysis These tools allow statistics on program operation to be collected and analyzed. See Chapter 22.

Multiple Processes Multiple and independent processes simplify problems which require logically separate pieces of code to operate in parallel. See Chapter 23.

Windows The ability to have multiple, independent windows on the display allows many different processes or activities to be active on the screen at once. See Chapter 28.

Inspector The inspector is a display tool for examining complex data structures encountered during debugging. See Chapter 26.

These facilities are tightly integrated, so they know about and use each other, just as they can be used by user programs. For example, Masterscope uses the structural editor to make systematic changes. By combining the program analysis features of Masterscope with the features of the structural editor, large scale system changes can be made with a single command. For example, when the lowest-level interface of the Medley I/O system was changed to a new format, the entire edit was made by a single call to Masterscope of the form `EDIT WHERE ANY CALLS '(BIN BOUT ...).` [Burton et al., 1980] This caused Masterscope to invoke the editor at each point in the system where any of the functions in the list `'(BIN BOUT ...)` were called. This ensured that no functions used in input or output were overlooked during the modification.

Philosophy

Medley's extensive environmental support has developed over the years to support a particular style of programming called "exploratory programming" [Sheil, 1983]. For many complex programming problems, the task of program creation is *not* simply one of writing a program to fulfill specifications. Instead, it is a matter of exploring the problem (trying out various solutions expressed as partial programs) until one finds a good solution (or sometimes, any solution at all!). Such programs are by nature evolutionary; they are transformed over time from one realization to another in response to a growing understanding of the problem. This point of view has lead to an emphasis on having the tools available to analyze, alter, and test programs easily. One important aspect of this is that the tools be designed to work together in an integrated fashion, so that knowledge about the user's programs, once gained, is available throughout the environment.

The development of programming tools to support exploratory programming is itself an exploration. No one knows all the tools that will eventually be found useful, and not all programmers want all of the tools to behave the same way. In response to this diversity, Interlisp has been shaped, by its implementors and by its users, to be easily extensible in several different ways. First, there are many places in the system where its behavior can be adjusted by the user. One way that this can be done is by changing the value of various "flags" or variables whose values are examined by system code to enable or suppress certain behavior. The other is where the user can provide functions or other behavioral specifications of what is to happen in certain contexts. For example, the format used for each type of list structure when it is printed by the pretty-printer is determined by specifications that are found on the list `PRETTYPRINTMACROS`. Thus, this format can be changed for a given type simply by putting a printing specification for it on that list.

Another way in which users can affect Medley's behavior is by redefining or changing system functions. The "Advise" capability, for instance, lets you modify the operation of virtually any function in the system by wrapping code "around" the selected function. (This same philosophy extends to breaking and tracing, so almost any function in the system can be broken or traced.) Since

the entire system is implemented in Lisp, there are few places where the system's behavior depends on anything that you can't modify (such as a low level system implementation language).

While these techniques provide a fair amount of tailorability, there's a price: Medley is complex. There are many flags, parameters, and controls that affect its behavior. Because of this complexity, Interlisp tends to be more comfortable for experts, rather than casual users. Beginning users of Interlisp should depend on the default settings of parameters until they learn what dimensions of flexibility are available. At that point, they can begin to "tune" the system to their preferences.

Appropriately enough, even Medley's underlying philosophy was itself discovered during Medley's development, rather than laid out beforehand. The Medley environment and its interactive style were first analyzed in Sandewall's excellent paper [Sandewall, 1978]. The notion of "exploratory programming" and the genesis of the Interlisp programming tools in terms of the characteristic demands of this style of programming was developed in [Sheil, 1983]. The evolution and structure of the Interlisp programming environment are discussed in greater depth in [Teitelman & Masinter, 1981].

How to Use this Manual

This document is a reference manual, not a primer. We have tried to provide a manual that is complete, and that lets you find particular items as easily as possible. Sometimes, these goals have been achieved at the expense of simplicity. For example, many functions have a number of arguments that are rarely used. In the interest of providing a complete reference, these arguments are fully explained, even though you will normally let them default. There is a lot of information in this manual that is of interest only to experts.

Do not try to read straight through this manual, like a novel. In general, the chapters are organized with overview explanations and the most useful functions at the beginning of the chapter, and implementation details towards the end. If you are interested in becoming acquainted with Medley, we urge you to work through *An Introduction to Medley* before attempting this manual.

A few comments about the notational conventions used in this manual:

Lisp object notation: All Interlisp objects in this manual are printed in the same font: Functions (`AND`, `PLUS`, `DEFINEQ`, `LOAD`); Variables (`MAX.INTEGER`, `FILELST`, `DFNFLG`); and arbitrary Interlisp expressions: `(PLUS 2 3)`, `(PROG ((A 1)) ...)`, etc.

Case is significant: *In Interlisp, upper and lower case is significant.* The variable `FOO` is not the same as the variable `foo` or the variable `Foo`. By convention, most Interlisp system functions and variables are all uppercase, but users are free to use upper and lower case for their own functions and variables as they wish.

One exception to the case-significance rule is provided by the CLISP facility, which lets you type iterative statements and record operations in either all uppercase or all lowercase letters: (`for X`

from 1 to 5 ...) is the same as (FOR X FROM 1 TO 5 ...). The few situations where this is the case are explicitly mentioned in the manual. Generally, assume that case is significant.

This manual contains a large number of descriptions of functions, variables, commands, etc, which are printed in the following standard format:

(**FOO** *BAR BAZ*)   [Function]

> This is a description for the function named FOO. FOO has two arguments, *BAR* and *BAZ*. Some system functions have extra optional arguments that are not documented and should not be used. These extra arguments are indicated by "—".
>
> The descriptor [Function] indicates that this is a function, rather than a [Variable], [Macro], etc. For function definitions only, this can also indicate whether the function takes a fixed or variable number of arguments, and whether the arguments are evaluated or not. [Function] indicates a lambda spread function (fixed number of arguments, evaluated), the most common type.

References

[Burton, et al., 1980]	Burton, R. R., L. M. Masinter, A. Bell, D. G. Bobrow, W. S. Haugeland, R.M. Kaplan and B.A. Sheil, "Interlisp-D: Overview and Status" — in [Sheil & Masinter, 1983].
[Sandewall, 1978]	Sandewall, Erik, "Programming in the Interactive Environmnet: The LISP Experience" — *ACM Computing Surveys*, vol 10, no 1, pp 35-72, (March 1978).
[Sheil, 1983]	Sheil, B.A., "Environments for Exploratory Programming" — *Datamation*, (February, 1983) — also in [Sheil & Masinter, 1983].
[Sheil & Masinter, 1983]	Sheil, B.A. and L. M. Masinter, "Papers on Interlisp-D", Xerox PARC Technical Report CIS-5 (Revised), (January, 1983).
[Teitelman & Masinter, 1981]	Teitelman, W. and L. M. Masinter, "The Interlisp Programming Environment" — *Computer*, vol 14, no 4, pp 25-34, (April 1981) — also in [Sheil & Masinter, 1983].

A litatom (for "literal atom") is an object that conceptually consists of a print name, a value, a function definition, and a property list. Litatoms are also known as "symbols" in Common Lisp. For clarity, we will use the term "symbol".

A symbol is read as any string of non-delimiting characters that cannot be interpreted as a number. The syntactic characters that delimit symbols are called "separator" or "break" characters (see Chapter 25) and normally are space, end-of-line, line-feed, left parenthesis `(`, right parenthesis `)`, double quote `"`, left square bracket `[`, and right square bracket `]`. However, any character may be included in a symbol by preceding it with the character `%`. Here are some examples of symbols:

```
A wxyz 23SKIDDOO %]
Long% Litatom% With% Embedded% Spaces
```

(**LITATOM** *X*)   [Function]

> Returns `T` if *X* is a symbol, `NIL` otherwise. Note that a number is not a symbol.
>
> ```
> (LITATOM NIL) = T
> ```

(**ATOM** *X*)   [Function]

> Returns `T` if *X* is an atom (i.e., a symbol or a number) or `NIL` (e.g. `(ATOM NIL) = T`); otherwise returns `NIL`.
>
> **Warning:** `(ATOM X)` is `NIL` if X is an array, string, etc. In Common Lisp, the function `CL:ATOM` is defined equivalent to the Interlisp function `NLISTP`.

Each symbol has a print name, a string of characters that uniquely identifies that symbol: Those characters that are output when the symbol is printed using `PRIN1`, e.g., the print name of the symbol `ABC%(D` consists of the five characters `ABC(D`.

Symbols are unique: If two symbols print the same, they will always be `EQ`. Note that this is not true for strings, large integers, floating-point numbers, etc.; they all can print the same without being `EQ`. Thus, if `PACK` or `MKATOM` is given a list of characters corresponding to a symbol that already exists, they return a pointer to that symbol, and do not make a new symbol. Similarly, if the read program is given as input a sequence of characters for which a symbol already exists, it returns a pointer to that symbol.

Symbol names are limited to 255 characters. Attempting to create a larger symbol will cause an error: `Atom too long`.

Sometimes we'll refer to a "PRIN2-name". The `PRIN2`-name of a symbol is those characters output when it is printed using `PRIN2`. So the `PRIN2`-name of the symbol `ABC%(D` is the six characters `ABC%(D`. The `PRIN2`-name depends on what readtable is being used (see Chapter 25), since this determines where `%`s will be inserted. Many of the functions below allow either print names or `PRIN2`-names to be used, as specified by `FLG` and `RDTBL` arguments. If `FLG` is `NIL`, print names are used. Otherwise, `PRIN2`-names are used, computed with respect to the readtable `RDTBL` (or the current readtable, if `RDTBL = NIL`).

(**MKATOM** *X*) [Function]

Creates and returns a symbol whose print name is the name as that of the string *X* or, if *X* is not a string, the same as that of (MKSTRING X). Examples:

```
(MKATOM '(A B C)) => %(A% B% C%)
(MKATOM "1.5") => 1.5
```

Note that the last example returns a number, not a symbol. It is a deeply-ingrained feature of Interlisp that no symbol can have the print name of a number.

(**SUBATOM** *X N M*) [Function]

Returns a symbol made from the *N*th through *M*th characters of the print name of *X*. If *N* or *M* are negative, they specify positions counting backwards from the end of the print name. Equivalent to (MKATOM (SUBSTRING X N M)). Examples:

```
(SUBATOM "FOO1.5BAR" 4 6) => 1.5
(SUBATOM '(A B C) 2 -2) => A% B% C
```

(**PACK** *X*) [Function]

If *X* is a list of symbols, PACK returns a single symbol whose print name is the concatenation of the print names of the symbols in *X*. If the concatenated print name is the same as that of a number, PACK returns that number. For example:

```
(PACK '(A BC DEF G)) => ABCDEFG
(PACK '(1 3.4)) => 13.4
(PACK '(1 E -2)) => .01
```

Although *X* is usually a list of symbols, it can be a list of arbitrary objects. The value of PACK is still a single symbol whose print name is the concatenation of the print names of all the elements of *X*, e.g.,

```
(PACK '((A B) "CD")) => %(A% B%)CD
```

If *X* is not a list or NIL, PACK generates the error Illegal arg.

(**PACK*** *X$_1$ X$_2$... X$_N$*) [NoSpread Function]

Version of PACK that takes an arbitrary number of arguments, instead of a list. Examples:

```
(PACK* 'A 'BC 'DEF 'G => ABCDEFG
(PACK* 1 3.4)) => 13.4
```

(**GENSYM** *PREFIX* − − − −) [Function]

Returns a symbol of the form X*nnnn*, where X = *PREFIX* (or A if *PREFIX* is NIL) and *nnnn* is an integer. Thus, the first one generated is A0001, the second A0002, etc. The integer suffix is always at least four characters long, but it can grow beyond that. For example, the next symbol produced after A9999 would be A10000. GENSYM provides a way of generating symbols for various uses within the system .

Note: The Common Lisp function CL:GENSYM is not the same as Interlisp's GENSYM. Interlisp always creates interned symbols whereas CL:GENSYM creates uninterned symbols.

GENNUM [Variable]

The value of GENNUM, initially 0, determines the next GENSYM, e.g., if GENNUM is set to 23, (GENSYM) = A0024.

The term "gensym" is used to indicate a symbol that was produced by the function GENSYM. Symbols generated by GENSYM are the same as any other symbols: they have property lists, and can be given function definitions. The symbols are not guaranteed to be new. For example, if the user has previously created A0012, either by typing it in, or via PACK or GENSYM itself, then if GENNUM is set to 11, the next symbol returned by GENSYM will be the A0012 already in existence.

(**MAPATOMS** *FN*) [Function]

Applies *FN* (a function or lambda expression) to every symbol in the system. Returns NIL. For example:

    (MAPATOMS (FUNCTION (LAMBDA(X) (if (GETD X) then (PRINTX]

will print every symbol with a function definition.

Warning: Be careful if *FN* is a lambda expression or an interpreted function: since NOBIND is a symbol, it will eventually be passed as an argument. The first reference to that argument within the function will signal an error.

A way around this problem is to use a Common Lisp function, so that the Common Lisp interpreter will be invoked. It will treat the argument as local, not special and no error will be signaled. An alternative solution is to include the argument to the Interlisp function in a LOCALVARS declaration and then compile the function before passing it to MAPATOMS. This will significantly speed up MAPATOMS.

(**APROPOS** *STRING ALLFLG QUITFLG OUTPUT*) [Function]

APROPOS scans all symbols in the system for those which have *STRING* as a substring and prints them on the terminal along with a line for each relevant item defined for each selected symbol. Relevant items are:

- function definitions, for which only the arglist is printed
- dynamic variable values
- non-null property lists

PRINTLEVEL (see Chapter 25) is set to (3 . 5) when APROPOS is printing.

If *ALLFLG* is NIL, then symbols with no relevant items and "internal" symbols are omitted ("internal" currently means those symbols whose print name begins with a \ or those symbols produced by GENSYM). If *ALLFLG* is a function, it is used as a predicate on symbols selected by the substring match, with value NIL meaning to omit the symbol. If *ALLFLG* is any other non-NIL value, then no symbols are omitted.

Note: Unlike CL:APROPOS which lets you designate the package to search, APROPOS searches *all* packages.

Using Symbols as Variables

Symbols are commonly used as variable names. Each symbol has a "top level" value, which can be an arbitrary object. Symbols may also be given special variable bindings within PROGs or functions, which only exist for the duration of the function. When a symbol is evaluated, the "current" variable binding is returned. This is the most recent special variable binding, or the top-level binding if the symbol hasn't been rebound. SETQ is used to change the current binding. For more information on variable bindings in Interlisp, see Chapter 11.

A symbol whose top-level value is the symbol NOBIND is considered to have no value. If a symbol has no local bindings, and its top-level value is NOBIND, trying to evaluate it will cause an unbound-atom error. In addition, if a symbol's local binding is to NOBIND, trying to evaluate it will cause an error.

The symbols T and NIL always evaluate to themselves. Attempting to change the value of T or NIL with the functions below will generate the error; Attempt to set T or Attempt to set NIL.

The following functions (except BOUNDP) will also generate the error Arg not litatom, if not given a symbol.

(**BOUNDP** *VAR*)   [Function]

Returns T if *VAR* has a special variable binding, or if *VAR* has a top-level value other than NOBIND; otherwise NIL. That is, if X is a symbol, (EVAL X) will cause an Unbound atom error if and only if (BOUNDP X) returns NIL.

Note: The Interlisp interpreter has been modified so that it will generate an Unbound Variable error when it encounters any symbol bound to NOBIND. This is a change from previous releases that only signaled an error when a symbol had a top-level binding of NOBIND in addition to no dynamic binding.

(**SET** *VAR VALUE*)   [NoSpread Function]

Sets the "current" value of *VAR* to *VALUE*, and returns *VALUE*.

SET is a normal function, so both *VAR* and *VALUE* are evaluated before it is called. Thus, if the value of X is B, and value of Y is C, then (SET X Y) would result in B being set to C, and C being returned as the value of SET.

(**SETQ** *VAR VALUE*)   [NoSpread Function]

Like SET, but *VAR* is not evaluated, *VALUE* is. Thus, if the value of X is B and the value of Y is C, (SETQ X Y) would result in X (not B) being set to C, and C being returned.

Actually, neither argument is evaluated during the calling process. However, SETQ itself calls EVAL on its second argument. As a result, typing (SETQ VAR FORM) and SETQ (VAR FORM) to the Interlisp Executive are equivalent: in both cases VAR is not evaluated, and FORM is.

(**SETQQ** *VAR VALUE*)   [NoSpread Function]

Like SETQ, but neither argument is evaluated, e.g., (SETQQ X (A B C)) sets X to (A B C).

(**PSETQ** VAR_1 $VALUE_1$ ... VAR_N $VALUE_N$)   [Macro]

> Does a SETQ in parallel of VAR_1 (unevaluated) to $VALUE_1$, VAR_2 to $VALUE_2$, etc. All of the $VALUE_i$ terms are evaluated before any of the assignments. Therefore, (PSETQ A B B A) can be used to swap the values of the variables A and B.

(**GETTOPVAL** VAR)   [Function]

> Returns the top level value of VAR (even if NOBIND), regardless of any intervening local bindings.

(**SETTOPVAL** VAR $VALUE$)   [Function]

> Sets the top level value of VAR to $VALUE$, regardless of any intervening bindings, and returns $VALUE$.

(**GETATOMVAL** VAR)   [Function]

> Same as (GETTOPVAL VAR).

(**SETATOMVAL** VAR $VALUE$)   [Function]

> Same as SETTOPVAL.

Note: The compiler (see Chapter 18) treats variables somewhat differently from the interpreter, and you need to be aware of these differences when writing functions that will be compiled. For example, variable references in compiled code are not checked for NOBIND, so compiled code will not generate unbound-atom errors. In general, it is better to debug interpreted code, before compiling it for speed. The compiler offers some facilities to increase the efficiency of variable use in compiled functions: Global variables can be defined so that the entire stack is not searched at each variable reference. Local variables have bindings that are not visible outside the function, which reduces variable conflicts and makes variable lookup faster.

Function Definition Cells

Each symbol has a function-definition cell, which is accessed when that symbol is used as a function. This is described in detail in Chapter 10.

Property Lists

Each symbol has an associated property list, which allows a set of named objects to be associated with the symbol. A property list associates a name (known as a "property name" or "property") with an arbitrary object (the "property value" or "value"). Sometimes the phrase "to store on the property X" is used, meaning to place the indicated information on a property list under the property name X.

Property names are usually symbols or numbers, although no checks are made. However, the standard property list functions all use EQ to search for property names, so they may not work with non-atomic property names. The same object can be used as both a property name and a property value.

Many symbols in the system already have property lists, with properties used by the compiler, the break package, DWIM, etc. Be careful not to clobber such system properties. The variable SYSPROPS is a list of property names used by the system.

The functions below are used to manipulate the property lists of symbols. Except when indicated, they generate the error *ATM* is not a SYMBOL, if given an object that is not a symbol.

(**GETPROP** *ATM PROP*)   [Function]

Returns the property value for *PROP* from the property list of *ATM*. Returns NIL if *ATM* is not a symbol, or *PROP* is not found. GETPROP also returns NIL if there is an occurrence of *PROP* but the corresponding property value is NIL. This can be a source of program errors.

Note: GETPROP used to be called GETP.

(**PUTPROP** *ATM PROP VAL*)   [Function]

Puts the property *PROP* with value *VAL* on the property list of *ATM*. *VAL* replaces any previous value for the property *PROP* on this property list. Returns *VAL*.

(**ADDPROP** *ATM PROP NEW FLG*)   [Function]

Adds the value *NEW* to the list which is the value of property *PROP* on the property list of the *ATM*. If *FLG* is T, *NEW* is CONSed onto the front of the property value of *PROP*; otherweise, it is NCONCed on the end (using NCONC1). If *ATM* does not have a property *PROP*, or the value is not a list, then the effect is the same as (PUTPROP ATM PROP (LIST NEW)). ADDPROP returns the (new) property value.   Example:

```
←(PUTPROP 'POCKET 'CONTENTS NIL)
 (NIL)
←(ADDPROP 'POCKET 'CONTENTS 'COMB)
 (COMB)
←(ADDPROP 'POCKET 'CONTENTS 'WALLET)
 (COMB WALLET)
```

(**REMPROP** *ATM PROP*)   [Function]

Removes all occurrences of the property *PROP* (and its value) from the property list of *ATM*. Returns *PROP* if any were found (T if *PROP* is NIL), otherwise NIL.

(**CHANGEPROP** *X PROP1 PROP2*)   [Function]

Changes the property name of property *PROP1* to *PROP2* on the property list of *X* (but does not affect the value of the property). Returns *X*, unless *PROP1* is not found, in which case it returns NIL.

(**PROPNAMES** *ATM*)   [Function]

Returns a list of the property names on the property list of *ATM*.

(**DEFLIST** *L PROP*)   [Function]

Used to put values under the same property name on the property lists of several symbols. *L* is a list of two-element lists. The first element of each is a symbol, and the second element is the property vqalue of the property *PROP*. Returns NIL. For example:

```
(DEFLIST '((FOO MA)(BAR CA)(BAZ RI)) 'STATE)
```

puts MA on FOO's STATE property, CA on BAR's STATE property, and RI on BAZ's STATE property.

Property lists are conventionally implemented as lists of the form

$$(NAME_1 \ VALUE_1 \ NAME_2 \ VALUE_2...)$$

although the user can store anything as the property list of a symbol. However, thge functions which manipulate property lists observe this convention by searching down the property lists two CDRs at a time. Most of these functions also generate the error Arg not litatom if given an argument which is not a symbol, so they cannot be used directly on lists. (LISTPUT, LISTPUT1, LISTGET, and LISTGET1 are functions similar to PUTPROP and GETPROP that work directly on lists (see Chapter 3) . The property lists of symbols can be directly accessed with the following functions.

(**GETPROPLIST** *ATM*)   [Function]

Returns the property list of *ATM*.

(**SETPROPLIST** *ATM LST*)   [Function]

If *ATM* is a symbol, sets the property list of *ATM* to be *LST*, and returns *LST* as its value.

(**GETLIS** *X PROPS*)   [Function]

Searches the property list of *X*, and returns the property list as of the first property on *PROPS* that it finds. For example:

```
←(GETPROPLIST 'X)
 (PROP1 A PROP3 B A C)
←(GETLIS 'X '(PROP2 PROP3))
 (PROP3 B A C)
```

Returns NIL if no element on props is found. *X* can also be a list itself, in which case it is searched as described above. If *X* is not a symbol or a list, returns NIL.

(**REMPROPLIST** *ATM PROPS*)   [Function]

Removes all occurrences of all properties on the list *PROPS* (and their corresponding property values) from the property list of *ATM*. Returns NIL.

Print Names

The term "print name" has an extended meaning: The characters that are output when *any object* is printed. In Medley, all objects have print names, although only symbols and strings have their print names explicitly stored. Symbol print names are limited to 255 characters.

This section describes a set of functions that can be used to access and manipulate the print names of any object, though they are primarily used with the print names of symbols. In Medley, print functions qualify symbol names with a package prefix if the symbol is not accessible in the current package. The exception is Interlisp's PRIN1, which does not include a package prefix.

The print name of an object is those characters that are output when the object is printed using PRIN1, e.g., the print name of the list (A B "C") consists of the seven characters (A B C) (two of the characters are spaces).

The PRIN2-name of an object is those characters output when the object is printed using PRIN2. Thus the PRIN2-name of the list (A B "C") is the 9 characters (A B "C") (including the two spaces). The PRIN2-name depends on what readtable is being used (see Chapter 25), since this determines where %s will be inserted. Many of the functions below allow either print names of PRIN2-names to be used, as specified by FLG and RDTBL arguments. If FLG is NIL, print names are used. Otherwise, PRIN2-names are used, computed with respect to the readtable RDTBL (or the current readtable, if RDTBL = NIL).

The print name of an integer depends on the setting of RADIX (see Chapter 25). The functions described in this section (UNPACK, NCHARS, etc.) define the print name of an integer as though the radix was 10, so that (PACK (UNPACK 'X9)) will always be X9 (and not X11, if RADIX is set to 8). However, integers will still be printed by PRIN1 using the current radix. The user can force these functions to use print names in the current radix by changing the setting of the variable PRXFLG (see Chapter 25).

(**CL:SYMBOL-NAME** *SYM*)   [Common Lisp Function]

> Returns a string displaced to the *SYM* print name. Strings returned from CL:SYMBOL-NAME may be destructively modified without affecting *SYM*'s print name.

(**NCHARS** *X FLG RDTBL*)   [Function]

> Returns the number of characters in the print name of *X*. If *FLG* = T, the PRIN2-name is used. Examples:
>
>   (NCHARS 'ABC) => 3
>   (NCHARS "ABC" T) => 5
>
> NCHARS works most efficiently on symbols and strings, but can be given any object.

(**NTHCHAR** *X N FLG RDTBL*)   [Function]

> Returns *X*, if *X* is a tail of the list Y; otherwise NIL. *X* is a tail of Y if it is EQ to 0 or more CDRs of Y.
>
>   (NTHCHAR 'ABC 2)   =>   B
>   (NTHCHAR 15.6 2)   =>   5
>   (NTHCHAR 'ABC%(D -3 T)   =>   %%
>   (NTHCHAR "ABC" 2)   =>   B
>   (NTHCHAR "ABC" 2 T)   =>   A
>
> NTHCAR and NCHARS work much faster on objects that actually have an internal representation of their print name, i.e., symbols and strings, than they do on numbers and lists, since they don't have to simulate printing.

(**L-CASE** *X FLG*)   [Function]

> Returns a lowercase version of *X*. If *FLG* is T, the first letter is capitalized. If *X* is a string, the value of L-CASE is also a string. If *X* is a list, L-CASE returns a new list in which L-CASE is computed for each corresponding element and non-NIL tail of the original list. Examples:
>
>   (L-CASE 'FOO) => foo
>   (L-CASE 'FOO T) => Foo
>   (L-CASE "FILE NOT FOUND" T) => "File not found"

```
(L-CASE '(JANUARY FEBRUARY (MARCH "APRIL")) T) =>
    '(January February (March "April"))
```

(**U-CASE** *X*)   [Function]

       Like L-CASE, but returns the uppercase version of *X*.

(**U-CASEP** *X*)   [Function]

       Returns T if *X* contains no lowercase letters; NIL otherwise.

Characters and Character Codes

Characters are represented 3 different ways in Medley. In Interlisp they are single-character symbols or integer character codes. In Common Lisp they are instances of the CHARACTER datatype. In general Interlisp character functions don't accept Common Lisp characters and vice versa. The only exceptions are Interlisp string-manipulation functions that accept "string or symbol" types as arguments.

You can convert between Interlisp and Common Lisp characaters by using the functions CL:CODE-CHAR, CL:CHAR-CODE, and CHARCODE (see below).

Medley uses the 16-bit NS character set, described in the document Character Code Standard (Xerox System Integration Standards, XSIS 058404, April 1984). Legal character codes range from 0 to 65535. The NS (Network Systems) character encoding encompasses a much wider set of available characters than the 8-bit character standards (such as ASCII), including characters comprising many foreign alphabets and special symbols. For instance, Medley supports the display and printing of the following:

- Le système d'information Medley est remarqueablement polyglotte

- Das Medley Kommunikationssystem bietet merkwürdige multilinguale Nutzmöglichkeiten

- $M \subseteq \Box$ [w] $\Leftrightarrow \forall$ v with Rwv: $M \subseteq$ [v]

These characters can be used in strings, symbol print names, symbolic files, or anywhere else 8-bit characters could be used. All of the standard string and print name functions (RPLSTRING, GNC, NCHARS, STRPOS, etc.) accept symbols and strings containing NS characters. For example:

```
←(STRPOS "char""this is an 8-bit character string")
   18
←(STRPOS "char""celui-ci comports des characteres NS")
   23
```

In almost all cases, a program does not have to distinguish between NS characters or 8-bit characters. The exception to this rule is the handling of input/output operations (see Chapter 25).

The function CHARCODE (see below) provides a simple way to create individual NS character codes. The VirtualKeyboards library module provides a set of virtual keyboards that allows keyboard or mouse entry of NS characters.

(**PACKC** *X*)   [Function]

       Like PACK except *X* is a list of character codes. For example,

```
(PACKC '(70 79 79)) => FOO
```

(**CHCON** *X FLG RDTBL*) [Function]

> Like UNPACK, but returns the print name of *X* as a list of character codes. If *FLG* = T, the PRIN2-name is used. For example:

>> (CHCON 'FOO) => (70 79 79)

(**DCHCON** *X SCRATCHLIST FLG RDTBL*) [Function]

> Like DUNPACK.

(**NTHCHARCODE** *X N FLG RDTBL*) [Function]

> Like NTHCHAR, but returns the character code of the *N*th character of the print name of *X*. If *N* is negative, it is interpreted as a count backwards from the end of *X*. If the absolute value of *N* is greater than the number of characters in *X*, or 0, then the value of NTHCHARCODE is NIL.

> If *FLG* is T, then the PRIN2-name of *X* is used, computed with respect to the readtable.

(**CHCON1** *X*) [Function]

> Returns the character code of the first character of the print name of *X*; equal to (NTHCHARCODE X 1).

(**CHARACTER** *N*) [Function]

> *N* is a character code. Returns the symbol having the corresponding single character as its print name.

>> (CHARACTER 70) => F

(**FCHARACTER** *N*) [Function]

> Fast version of CHARACTER that compiles open.

The following function makes it possible to gain the efficiency that comes from dealinig with character codes without losing the symbolic advantages of character symbols.

(**CHARCODE** *CHAR*) [Function]

> Returns the character code specified by *CHAR* (unevaluated). If *CHAR* is a one-character symbol or string, the corresponding character code is simply returned. Thus, (CHARCODE A) is 65, (CHARCODE 0) is 48. If *CHAR* is a multi-character symbol or string, it specifies a character code as described below. If *CHAR* is NIL, CHARCODE simply returns NIL. Finally, if *CHAR* is a list structure, the value is a copy of *CHAR* with all the leaves replaced by the corresponding character codes. For instance, (CHARCODE (A (B C))) => (65 (66 67)).

> If a character is specified by a multi-character symbol or string, CHARCODE interprets it as follows:

> CR, SPACE, etc.

The variable `CHARACTERNAMES` contains an association list mapping special symbols to character codes. Among the characters defined this way are CR (`13`), LF (10), SPACE or SP (32), ESCAPE or ESC (27), BELL (7), BS (8), TAB (9), NULL (0), and DEL (127). The symbol EOL maps into the appropriate end-of-line character code in the different Interlisp implementations (31 in Interlisp-10, 13 in Interlisp-D, 10 in Interlisp-VAX). Examples:

```
(CHARCODE SPACE)  =>  32
(CHARCODE CR)  =>  13
```

CHARSET, CHARNUM, CHARSET-CHARNUM

If the character specification is a symbol or string of the form `CHARSET`, `CHARNUM`, or `CHARSET-CHARNUM`, the character code for the character number `CHARNUM` in the character set `CHARSET` is returned.

The 16-bit NS character encoding is divided into a large number of "character sets". Each 16-bit character can be decoded into a character set (an integer from 0 to 254 inclusive) and a character number (also an integer from 0 to 254 inclusive). `CHARSET` is either an octal number, or a symbol in the association list `CHARACTERSETNAMES` (which defines the character sets for `GREEK`, `CYRILLIC`, etc.).

`CHARNUM` is either an octal number, a single-character symbol, or a symbol from the association list `CHARACTERNAMES`. If `CHARNUM` is a single-digit number, it is interpreted as the character "2", rather than as the octal number 2. Examples:

```
(CHARCODE 12,6)  =>  2566
(CHARCODE 12,SPACE)  =>  2592
(CHARCODE GREEK,A) => 9793
```

↑CHARSPEC (control chars)

If the character specification is a symbol or string of one of the forms above, preceded by the character ↑, this indicates a "control character," derived from the normal character code by clearing the seventh bit of the character code (normally set). Examples:

```
(CHARCODE ↑A)  =>  1
(CHARCODE ↑GREEK,A)  =>  9729
```

#CHARSPEC (meta chars)

If the character specification is a symbol or string of one of the forms above, preceded by the charactger #, this indicates a meta character, derived from the normal character code by setting the eighth bit of the character code (normally cleared). ↑ and # can both be set at once. Examples:

```
(CHARCODE #A)  =>  193
(CHARCODE #↑GREEK,A)  =>  9857
```

A `CHARCODE` form can be used wherever a structure of character codes would be appropriate. For example:

```
(FMEMB (NTHCHARCODE X 1)(CHARCODE (CR LF SPACE ↑A)))
(EQ (READCCODE FOO)(CHARCODE GREEK,A))
```

There is a macro for CHARCODE which causes the character-code structure to be constructed at compile-time. Thus, the compiled code for these examples is exactly as efficient as the less readable:

```
(FMEMB (NTHCHARCODE X 1)(QUOTE (13 10 32 1)))
(EQ (READCCODE FOO)9793)
```

(**CL:CHAR-CODE** *CHAR*)   [Common Lisp Function]

Returns the Interlisp character code of *CHAR*. Use to convert a Common Lisp character to an Interlisp character code.

(**CL:CODE-CHAR** *N*)   [Common Lisp Function]

Returns a character with the given non-negative integer *N* code. Returns NIL if no character is possible with *N*. Use to convert an Interlisp character code to a Common Lisp character.

(**SELCHARQ** *E CLAUSE$_1$... CLAUSE$_N$ DEFAULT*)   [Function]

Lets you branch one of several ways, based on the character code *E*. The first item in each *CLAUSE$_N$* is a character code or list of character codes, given in the form CHARCODE would accept. If the value of *E* is a character code or NIL, and it is EQ or MEMB to the result of applying CHARCODE to the first element of a clause, the remaining forms of that clause are evaluated. Otherwise, the default is evaluated.

Thus

```
(SELCHARQ (BIN FOO))
      ((SPACE TAB)(FUM))
      ((↑D NIL)(BAR))
      (a (BAZ))
      (ZIP)))
```

is exactly equivalent to

```
(SELECTQ (BIN FOO))
      ((32 9)(FUM))
      ((4 NIL)(BAR))
      (97 (BAZ))
      (ZIP)))
```

If (BIN FOO) returned 32 (the SPACE character), the function FUM would be called.

[This page intentionally left blank]

One of the most useful datatypes in Lisp is the list cell, a data structure that contains pointers to two other objects, called the CAR and the CDR of the list cell. You can build very complicated structures out of list cells, including lattices and trees, but most often they're used to represent simple linear lists of objects.

The following functions are used to manipulate individual list cells:

(**CONS** *X Y*) [Function]

> CONS is the primary list construction function. It creates and returns a new list cell containing pointers to *X* and *Y*. If *Y* is a list, this returns a list with *X* added at the beginning of *Y*.

(**LISTP** *X*) [Function]

> Returns *X* if *X* is a list cell, e.g., something created by CONS; NIL otherwise.

>> (LISTP NIL) = NIL

(**NLISTP** *X*) [Function]

> The same as (NOT (LISTP *X*)). Returns T if *X* is not a list cell, NIL otherwise. However, (NLISTP NIL) = T

(**CAR** *X*) [Function]

> Returns the first element of the list *X*. CAR of NIL is always NIL. For all other nonlists (e.g., symbols, numbers, etc.), the value returned is controlled by CAR/CDRERR (below).

(**CDR** *X*) [Function]

> Returns all but the first element of the list *X*. CDR of NIL is always NIL. The value of CDR for other nonlists is controlled by CAR/CDRERR (below).

CAR/CDRERR [Variable]

> The variable CAR/CDRERR controls the behavior of CAR and CDR when they are passed non-lists (other than NIL).

> If CAR/CDRERR = NIL (the current default), then CAR or CDR of a non-list (other than NIL) return the string "{car of non-list}" or "{cdr of non-list}". If CAR/CDRERR = T, then CAR and CDR of a non-list (other than NIL) causes an error.

> If CAR/CDRERR = ONCE, then CAR or CDR of a string causes an error, but CAR or CDR of anything else returns the string "{car of non-list}" or "{cdr of non-list}" as above. This catches loops which repeatedly take CAR or CDR of an object, but it allows one-time errors to pass undetected.

> If CAR/CDRERR = CDR, then CAR of a non-list returns "{car of non-list}" as above, but CDR of a non-list causes an error. This setting is based on the observation that

nearly all infinite loops involving non-lists occur from taking CDRs, but a fair amount of careless code takes CAR of something it has not tested to be a list.

(**CAAR** *X*) (**CADR** *X*) (**CDDR** *X*) etc. [Function]

Often, combinations of CAR and CDR are used to extract parts of complex list structures. Functions of the form C...R may be used for some of these combinations:

```
(CAAR X) ==> (CAR (CAR X))
(CADR X) ==> (CAR (CDR X))
(CDDDDR X) ==> (CDR (CDR (CDR (CDR X))))
```

All 30 combinations of nested CARs and CDRs up to 4 deep are included in the system.

(**RPLACD** *X Y*) [Function]

Replaces the CDR of the list cell *X* with *Y*. This physically changes the internal structure of *X*, as opposed to CONS, which creates a new list cell. You can make a circular list by using RPLACD to place a pointer to the beginning of a list at the end of the list.

The value of RPLACD is *X*. An attempt to RPLACD NIL will cause an error, Attempt to RPLACD NIL (except for (RPLACD NIL NIL)). An attempt to RPLACD any other non-list will cause an error, Arg not list.

(**RPLACA** *X Y*) [Function]

Like RPLACD, but replaces the CAR of *X* with *Y*. The value of RPLACA is *X*. An attempt to RPLACA NIL will cause an error, Attempt to RPLACA NIL, (except for (RPLACA NIL NIL)). An attempt to RPLACA any other non-list will cause an error, Arg not list.

(**RPLNODE** *X A D*) [Function]

Performs (RPLACA *X A*), (RPLACD *X D*), and returns *X*.

(**RPLNODE2** *X Y*) [Function]

Performs (RPLACA *X* (CAR *Y*)), (RPLACD *X* (CDR *Y*)) and returns *X*.

(**FRPLACD** *X Y*) [Function]
(**FRPLACA** *X Y*) [Function]
(**FRPLNODE** *X A D*) [Function]
(**FRPLNODE2** *X Y*) [Function]

Faster versions of RPLACD, etc.

Usually, you don't use list cells alone, but in structures called "lists". A list is represented by a list cell whose CAR is the first element of the list, and whose CDR is the rest of the list. That's normally another list cell (with another element of the list) or the "empty list," NIL, marking the list's end. List elements may be any Lisp objects, including other lists.

You type in a list as a sequence of Lisp data objects (symbols, numbers, other lists, etc.) enclosed in parentheses or brackets. Note that () is read as the symbol NIL.

Sometimes, you won't want your list to end in `NIL`, but just with the final element. To indicate that, type a period (with spaces on both sides) in front of the final element. This makes `CDR` of the list's final cell be the element immediately following the period, e.g. `(A . B)` or `(A B C . D)`. Note that a list needn't end in `NIL`. It is simply a structure composed of one or more list cells. The input sequence `(A B C . NIL)` is equivalent to `(A B C)`, and `(A B . (C D))` is equivalent to `(A B C D)`. Note, however, that `(A B . C D)` will create a list containing the five symbols `A`, `B`, `%.`, `C`, and `D`.

Lists are printed by printing a left parenthesis, and then printing the first element of the list, a space, the second element, etc., until the final list cell is reached. The individual elements of a list are printed by `PRIN1`, if the list is being printed by `PRIN1`, and by `PRIN2` if the list is being printed by `PRINT` or `PRIN2`. Lists are considered to terminate when `CDR` of some node is not a list. If `CDR` of this terminal node is `NIL` (the usual case), `CAR` of the last node is printed followed by a right parenthesis. If `CDR` of the terminal node is *not* `NIL`, `CAR` of the last node is printed, followed by a space, a period, another space, `CDR` of the last node, and the right parenthesis. A list input as `(A B C . NIL)` will print as `(A B C)`, and a list input as `(A B . (C D))` will print as `(A B C D)`. `PRINTLEVEL` affects the printing of lists (see the `PRINTLEVEL` section of Chapter 25), and that carriage returns may be inserted where dictated by `LINELENGTH` (see the Output Functions section of Chapter 25).

> Note: Be careful when testing the equality of list structures. `EQ` will be true only when the two lists are the *exact* same list. For example,
>
> ```
> ← (SETQ A '(1 2))
>   (1 2)
> ← (SETQ B A)
>   (1 2)
> ← (EQ A B)
>   T
> ←(SETQ C '(1 2))
>   (1 2)
> ←(EQ A C)
>   NIL
> ←(EQUAL A C)
>   T
> ```
>
> In the example above, the values of A and B are the exact same list, so they are `EQ`. However, the value of C is a totally different list, although it happens to have the same elements. `EQUAL` should be used to compare the elements of two lists. In general, one should notice whether list manipulation functions use `EQ` or `EQUAL` for comparing lists. This is a frequent source of errors.

Creating Lists

(**LIST** X_1 X_2 ... X_N)   [NoSpread Function]

> Returns a list of its arguments, e.g.
>
> ```
> (LIST 'A 'B '(C D))  =>  (A B (C D))
> ```

(**LIST*** X_1 X_2 ... X_N)   [NoSpread Function]

> Returns a list of its arguments, using the last argument for the tail of the list. This is like an iterated `CONS`: `(LIST* A B C) == (CONS A (CONS B C))`. For example,

```
(LIST* 'A 'B 'C)  =>  (A B . C)
(LIST* 'A 'B '(C D))  =>  (A B C D)
```

(**APPEND** X_1 X_2 ... X_N)   [NoSpread Function]

Copies the top level of the list X_1 and appends this to a copy of the top level of the list X_2 appended to . . . appended to X_N, e.g.,

```
(APPEND '(A B) '(C D E) '(F G))  =>  (A B C D E F G)
```

Only the first $N-1$ lists are copied. However $N = 1$ is treated specially; (APPEND X) copies the top level of a single list. To copy a list to all levels, use COPY.

The following examples illustrate the treatment of non-lists:

```
(APPEND '(A B C) 'D)  =>  (A B C . D)
(APPEND 'A '(B C D))  =>  (B C D)
(APPEND '(A B C . D) '(E F G))  =>  (A B C E F G)
(APPEND '(A B C . D))  =>  (A B C . D)
```

(**NCONC** X_1 X_2 ... X_N)   [NoSpread Function]

Returns the same value as APPEND, but modifies the list structure of X_1 ... X_{n-1}.

NCONC cannot change NIL to a list:

```
←(SETQ FOO NIL)
   NIL
←(NCONC FOO '(A B C))
   (A B C)
←FOO
   NIL
```

Although the value of the NCONC is (A B C), FOO has *not* been changed. The "problem" is that while it is possible to alter list structure with RPLACA and RPLACD, there is no way to change the non-list NIL to a list.

(**NCONC1** LST X)   [Function]

Adds X to the end of LST: (NCONC LST (LIST X))

(**ATTACH** X L)   [Function]

"Attaches" X to the front of L by doing a RPLACA and RPLACD. The value is EQUAL to (CONS X L), but EQ to L, which it physically changes (except if L is NIL). (ATTACH X NIL) is the same as (CONS X NIL). Otherwise, if L is not a list, an error is generated, Arg not list.

(**MKLIST** X)   [Function]

"Make List." If X is a list or NIL, returns X; Otherwise, returns (LIST X).

Building Lists From Left to Right

(**TCONC** *PTR X*)   [Function]

> TCONC is similar to NCONC1; it is useful for building a list by adding elements one at a time at the end. Unlike NCONC1, TCONC does not have to search to the end of the list each time it is called. Instead, it keeps a pointer to the end of the list being assembled, and updates this pointer after each call. This can be considerably faster for long lists. The cost is an extra list cell, *PTR*. (CAR *PTR*) is the list being assembled, (CDR *PTR*) is (LAST (CAR *PTR*)). TCONC returns *PTR*, with its CAR and CDR appropriately modified.
>
> *PTR* can be initialized in two ways. If *PTR* is NIL, TCONC will create and return a *PTR*. In this case, the program must set some variable to the value of the first call to TCONC. After that, it is unnecessary to reset the variable, since TCONC physically changes its value. Example:
>
> ```
>     ←(SETQ FOO (TCONC NIL 1))
>        ((1) 1)
>     ←(for I from 2 to 5 do (TCONC FOO I))
>        NIL
>     ←FOO
>        ((1 2 3 4 5) 5)
> ```
>
> If *PTR* is initially (NIL), the value of TCONC is the same as for *PTR* = NIL. but TCONC changes *PTR*. This method allows the program to initialize the TCONC variable before adding any elements to the list. Example:
>
> ```
>     ←(SETQ FOO (CONS))
>        (NIL)
>     ←(for I from 1 to 5 do (TCONC FOO I))
>        NIL
>     ←FOO
>        ((1 2 3 4 5) 5)
> ```

(**LCONC** *PTR X*)   [Function]

> Where TCONC is used to add *elements* at the end of a list, LCONC is used for building a list by adding *lists* at the end, i.e., it is similar to NCONC instead of NCONC1. Example:
>
> ```
>     ←(SETQ FOO (CONS))
>        (NIL)
>     ←(LCONC FOO '(1 2))
>        ((1 2) 2)
>     ←(LCONC FOO '(3 4 5))
>        ((1 2 3 4 5) 5)
>     ←(LCONC FOO NIL)
>        ((1 2 3 4 5) 5)
> ```
>
> LCONC uses the same pointer conventions as TCONC for eliminating searching to the end of the list, so that the same pointer can be given to TCONC and LCONC interchangeably. Therefore, continuing from above,
>
> ```
>     ←(TCONC FOO NIL)
>        ((1 2 3 4 5 NIL) NIL)
> ```

```
←(TCONC FOO '(3 4 5))
  ((1 2 3 4 5 NIL (3 4 5)) (3 4 5))
```

The functions DOCOLLECT and ENDCOLLECT also let you build lists from left-to-right like TCONC, but without the overhead of an extra list cell. The list is kept as a circular list. DOCOLLECT adds items; ENDCOLLECT replaces the tail with its second argument, and returns the full list.

(**DOCOLLECT** *ITEM LST*)   [Function]

> "Adds" *ITEM* to the end of *LST*. Returns the new circular list. Note that *LST* is modified, but it is not EQ to the new list. The new list should be stored and used as *LST* to the next call to DOCOLLECT.

(**ENDCOLLECT** *LST TAIL*)   [Function]

> Takes *LST*, a list returned by DOCOLLECT, and returns it as a non-circular list, adding *TAIL* as the terminating CDR.

> Here is an example using DOCOLLECT and ENDCOLLECT. HPRINT is used to print the results because they are circular lists. Notice that FOO has to be set to the value of DOCOLLECT as each element is added.

```
←(SETQ FOO NIL]
  NIL
←(HPRINT (SETQ FOO (DOCOLLECT 1 FOO]
  ↑(1 . {1})
←(HPRINT (SETQ FOO (DOCOLLECT 2 FOO]
  ↑(2 1 . {1})
←(HPRINT (SETQ FOO (DOCOLLECT 3 FOO]
  ↑(3 1 2 . {1})
←(HPRINT (SETQ FOO (DOCOLLECT 4 FOO]
  ↑(4 1 2 3 . {1})
←(SETQ FOO (ENDCOLLECT FOO 5]
  (1 2 3 4 . 5)
```

The following two functions are useful when writing programs that reuse a scratch list to collect together some result(s) (both of these compile open):

(**SCRATCHLIST** *LST* $X_1 X_2 \ldots X_N$)   [NLambda NoSpread Function]

> SCRATCHLIST sets up a context in which the value of *LST* is used as a "scratch" list. The expressions X_1, X_2, ... X_N are evaluated in turn. During the course of evaluation, any value passed to ADDTOSCRATCHLIST will be saved, reusing CONS cells from the value of *LST*. If the value of *LST* is not long enough, new CONS cells will be added onto its end. If the value of *LST* is NIL, the entire value of SCRATCHLIST will be "new" (i.e., no CONS cells will be reused).

(**ADDTOSCRATCHLIST** *VALUE*)   [Function]

> For use under calls to SCRATCHLIST. *VALUE* is added on to the end of the list of things being collected by SCRATCHLIST. When SCRATCHLIST returns, its value is a list containing all of the things added by ADDTOSCRATCHLIST.

Copying Lists

(**COPY** *X*) [Function]

Creates and returns a copy of the list *X*. All levels of *X* are copied down to non-lists, so that if *X* contains arrays and strings, the copy of *X* will contain the same arrays and strings, not copies. COPY is recursive in the CAR direction only, so very long lists can be copied.

To copy just the *top level* of *X*, do (APPEND *X*).

(**COPYALL** *X*) [Function]

Like COPY, but it copies down to atoms. Arrays, hash-arrays, strings, user data types, etc., are all copied. Analagous to EQUALALL (see the Equality Predicates section of Chapter 9). This will not work if given a data structure with circular pointers; in this case, use HCOPYALL.

(**HCOPYALL** *X*) [Function]

Like COPYALL, but it will work even if the data structure contains circular pointers.

Extracting Tails of Lists

(**NTH** *X N*) [Function]

Returns the tail of *X* beginning with the *N*th element. Returns NIL if *X* has fewer than *N* elements. This is different from Common Lisp's NTH. Examples:

```
(NTH '(A B C D) 1)  =>  (A B C D)
(NTH '(A B C D) 3)  =>  (C D)
(NTH '(A B C D) 9)  =>  NIL
(NTH '(A . B) 2)  =>  B
```

For consistency, if *N* = 0, NTH returns (CONS NIL *X*):

```
(NTH '(A B) 0)  =>  (NIL A B)
```

(**FNTH** *X N*) [Function]

Faster version of NTH that terminates on a null-check.

(**LAST** *X*) [Function]

Returns the last list cell in the list *X*. Returns NIL if *X* is not a list. Examples:

```
(LAST '(A B C))   =>  (C)
(LAST '(A B . C)) =>  (B . C)
(LAST 'A)  =>  NIL
```

(**FLAST** *X*) [Function]

Faster version of LAST that terminates on a null-check.

(**NLEFT** *L N TAIL*) [Function]

NLEFT returns the tail of *L* that contains *N* more elements than *TAIL*. If *L* does not contain *N* more elements than *TAIL*, NLEFT returns NIL. If *TAIL* is NIL or not a tail of *L*, NLEFT

returns the last *N* list cells in *L*. NLEFT can be used to work backwards through a list. Example:

```
←(SETQ FOO '(A B C D E))
   (A B C D E)
←(NLEFT FOO 2)
   (D E)
←(NLEFT FOO 1 (CDDR FOO))
   (B C D E)
←(NLEFT FOO 3 (CDDR FOO))
   NIL
```

(**LASTN** *L* *N*) [Function]

Returns (CONS X Y), where Y is the last *N* elements of *L*, and X is the initial segment, e.g.,

```
(LASTN '(A B C D E) 2)  =>  ((A B C) D E)
(LASTN '(A B) 2)  =>  (NIL A B)
```

Returns NIL if *L* is not a list containing at least *N* elements.

(**TAILP** *X* *Y*) [Function]

Returns *X*, if *X* is a *tail* of the list *Y*; otherwise NIL. *X* is a tail of *Y* if it is EQ to 0 or more CDRs of *Y*.

Note: If *X* is EQ to 1 or more CDRs of *Y*, *X* is called a "proper tail."

Counting List Cells

(**LENGTH** *X*) [Function]

Returns the length of the list *X*, where "length" is defined as the number of CDRs required to reach a non-list. Examples:

```
(LENGTH '(A B C))  =>  3
(LENGTH '(A B C . D))  =>  3
(LENGTH 'A)  =>  0
```

(**FLENGTH** *X*) [Function]

Faster version of LENGTH that terminates on a null-check.

(**EQLENGTH** *X* *N*) [Function]

Equivalent to (EQUAL (LENGTH X) N), but more efficient, because EQLENGTH stops as soon as it knows that *X* is longer than *N*. EQLENGTH is safe to use on (possibly) circular lists, since it is "bounded" by *N*.

(**COUNT** *X*) [Function]

Returns the number of list cells in the list *X*. Thus, COUNT is like a LENGTH that goes to all levels. COUNT of a non-list is 0. Examples:

```
(COUNT '(A))  =>  1
(COUNT '(A . B))  =>  1
(COUNT '(A (B) C))  =>  4
```

In this last example, the value is 4 because the list (A X C) uses three list cells for any object *X*, and (B) uses another list cell.

(**COUNTDOWN** *X N*) [Function]

Counts the number of list cells in *X*, decrementing *N* for each one. Stops and returns *N* when it finishes counting, or when *N* reaches 0. COUNTDOWN can be used on circular structures since it is "bounded" by *N*. Examples:

```
(COUNTDOWN '(A) 100)  =>  99
(COUNTDOWN '(A . B) 100)  =>  99
(COUNTDOWN '(A (B) C) 100)  =>  96
(COUNTDOWN (DOCOLLECT 1 NIL) 100)  =>  0
```

(**EQUALN** *X Y DEPTH*) [Function]

Like EQUAL, for use with (possibly) circular structures. Whenever the depth of CAR recursion plus the depth of CDR recursion exceeds *DEPTH*, EQUALN does not search further along that chain, and returns the symbol ?. If recursion never exceeds *DEPTH*, EQUALN returns T if the expressions *X* and *Y* are EQUAL; otherwise NIL.

```
(EQUALN '(((A)) B) '(((Z)) B) 2)  =>  ?
(EQUALN '(((A)) B) '(((Z)) B) 3)  =>  NIL
(EQUALN '(((A)) B) '(((A)) B) 3)  =>  T
```

Set Operations

(**INTERSECTION** *X Y*) [Function]

Returns a list whose elements are members of both lists *X* and *Y* (using EQUAL to do compares).

Note that (INTERSECTION X X) gives a list of all members of X without duplicates.

(**UNION** *X Y*) [Function]

Returns a (new) list consisting of all elements included on either of the two original lists (using EQUAL to compare elements). It is more efficient for *X* to be the shorter list.

The value of UNION is *Y* with all elements of *X* not in *Y* CONSed on the front of it. Therefore, if an element appears twice in *Y*, it will appear twice in (UNION *X Y*). Since (UNION '(A) '(A A)) = (A A), while (UNION '(A A) '(A)) = (A), UNION is non-commutative.

(**LDIFFERENCE** *X Y*) [Function]

"List Difference." Returns a list of the elements in *X* that are not members of *Y* (using EQUAL to compare elements).

Note: If *X* and *Y* share no elements, LDIFFERENCE returns a copy of *X*.

(**LDIFF** *LST TAIL ADD*) [Function]

TAIL must be a tail of *LST*, i.e., EQ to the result of applying some number of CDRs to *LST*. (LDIFF *LST TAIL*) returns a list of all elements in *LST* up to *TAIL*.

If *ADD* is not NIL, the value of LDIFF is effectively (NCONC *ADD* (LDIFF *LST* *TAIL*)), i.e., the list difference is added at the end of *ADD*.

If *TAIL* is not a tail of *LST*, LDIFF generates an error, LDIFF: not a tail. LDIFF terminates on a null-check, so it will go into an infinite loop if *LST* is a circular list and *TAIL* is not a tail.

Example:

```
←(SETQ FOO '(A B C D E F))
    (A B C D E F)
←(CDDR FOO)
    (C D E F)
←(LDIFF FOO (CDDR FOO))
    (A B)
←(LDIFF FOO (CDDR FOO) '(1 2))
    (1 2 A B)
←(LDIFF FOO '(C D E F))
    LDIFF: not a tail
    (C D E F)
```

Note that the value of LDIFF is always new list structure unless *TAIL* = NIL, in which case the value is *LST* itself.

Searching Lists

(**MEMB** *X Y*)   [Function]

Determines if *X* is a member of the list *Y*. If there is an element of *Y* EQ to *X*, returns the tail of *Y* starting with that element. Otherwise, returns NIL. Examples:

```
(MEMB 'A '(A (W) C D))  =>  (A (W) C D)
(MEMB 'C '(A (W) C D))  =>  (C D)
(MEMB 'W '(A (W) C D))  =>  NIL
(MEMB '(W) '(A (W) C D))  =>  NIL
```

(**FMEMB** *X Y*)   [Function]

Faster version of MEMB that terminates on a null-check.

(**MEMBER** *X Y*)   [Function]

Identical to MEMB except that it uses EQUAL instead of EQ to check membership of *X* in *Y*. Examples:

```
(MEMBER 'C '(A (W) C D)) => (C D)
(MEMBER 'W '(A (W) C D)) => NIL
(MEMBER '(W) '(A (W) C D)) => ((W) C D)
```

(**EQMEMB** *X Y*)   [Function]

Returns T if either *X* is EQ to *Y*, or else *Y* is a list and *X* is an FMEMB of *Y*.

Substitution Functions

(**SUBST** *NEW OLD EXPR*)   [Function]

Returns the result of substituting *NEW* for all occurrences of *OLD* in the expression *EXPR*. Substitution occurs whenever *OLD* is EQUAL to CAR of some subexpression of *EXPR*, or when *OLD* is atomic and EQ to a non-NIL CDR of some subexpression of *EXPR*. For example:

```
(SUBST 'A 'B '(C B (X . B))) => (C A (X . A))
(SUBST 'A '(B C) '((B C) D B C)) => (A D B C) not (A D . A)
```

SUBST returns a copy of *EXPR* with the appropriate changes. Furthermore, if *NEW* is a list, it is copied at each substitution.

(**DSUBST** *NEW OLD EXPR*)   [Function]

Like SUBST, but it does not copy *EXPR*, but changes the list structure *EXPR* itself. Like SUBST, DSUBST substitutes with a copy of *NEW*. More efficient than SUBST.

(**LSUBST** *NEW OLD EXPR*)   [Function]

Like SUBST, but *NEW* is substituted as a segment of the list *EXPR* rather than as an element. For instance,

```
(LSUBST '(A B) 'Y '(X Y Z))  =>  (X A B Z)
```

If *NEW* is not a list, LSUBST returns a copy of *EXPR* with all *OLD*'s deleted:

```
(LSUBST NIL 'Y '(X Y Z))  =>  (X Z)
```

(**SUBLIS** *ALST EXPR FLG*)   [Function]

ALST is a list of pairs:

$$((OLD_1 . NEW_1) (OLD_2 . NEW_2) ... (OLD_N . NEW_N))$$

Each OLD_i is an atom. SUBLIS returns the result of substituting each NEW_i for the corresponding OLD_i in *EXPR*, e.g.,

```
(SUBLIS '((A . X) (C . Y)) '(A B C D))  =>  (X B Y D)
```

If *FLG* = NIL, new structure is created only if needed, so if there are no substitutions, the value is EQ to *EXPR*. If *FLG* = T, the value is always a copy of *EXPR*.

(**DSUBLIS** *ALST EXPR FLG*)   [Function]

Like SUBLIS, but it changes the list structure *EXPR* itself instead of copying it.

(**SUBPAIR** *OLD NEW EXPR FLG*)   [Function]

Like SUBLIS, but elements of *NEW* are substituted for corresponding atoms of *OLD* in *EXPR*, e.g.,

```
(SUBPAIR '(A C) '(X Y) '(A B C D))  =>  (X B Y D)
```

As with SUBLIS, new structure is created only if needed, or if *FLG* = T, e.g., if *FLG* = NIL and there are no substitutions, the value is EQ to *EXPR*.

If *OLD* ends in an atom other than NIL, the rest of the elements on *NEW* are substituted for that atom. For example, if *OLD* = (A B . C) and *NEW* = (U V X Y Z), U is substituted for A, V for B, and (X Y Z) for C. Similarly, if *OLD* itself is an atom (other than NIL), the entire list *NEW* is substituted for it. Examples:

```
(SUBPAIR '(A B . C) '(W X Y Z) '(C A B B Y)) => ((Y Z) W X X
Y)
```

SUBST, DSUBST, and LSUBST all substitute copies of the appropriate expression, whereas SUBLIS, and DSUBLIS, and SUBPAIR substitute the identical structure (unless *FLG* = T). For example:

```
← (SETQ FOO '(A B))
  (A B)
← (SETQ BAR '(X Y Z))
  (X Y Z)
← (DSUBLIS (LIST (CONS 'X FOO)) BAR)
  ((A B) Y Z)
← (DSUBLIS (LIST (CONS 'Y FOO)) BAR T)
  ((A B) (A B) Z)
← (EQ (CAR BAR) FOO)
  T
← (EQ (CADR BAR) FOO)
  NIL
```

Association Lists and Property Lists

It is often useful to associate a set of property names (*NAME1*, *NAME2*, etc.), with a set of property values (*VALUE1*, *VALUE2*, etc.). Two list structures commonly used to store such associations are called "property lists" and "association lists." A list in "association list" format is a list where each element is a call whose CAR is a property name, and whose CDR is the value:

```
( (NAME1 . VALUE1) (NAME2 . VALUE2) ...)
```

A list in "property list" format is a list where the first, third, etc. elements are the property names, and the second, forth, etc. elements are the associated values:

```
( NAME1 VALUE1 NAME2 VALUE2 ...)
```

Another data structure that offers some of the advantages of association lists and property lists is the hash array (see the first page of Chapter 6).

The functions below provide facilities for searching and changing lists in property list or association list format.

Note: Property lists are used in many Medley system datatypes. There are special functions that can be used to set and retrieve values from the property lists of symbols (see the Property Lists section of Chapter 2), from properties of windows (see the Window Properties section of Chapter 28), etc.

(**ASSOC** *KEY ALST*)   [Function]

ALST is a list of lists. ASSOC returns the first sublist of *ALST* whose CAR is EQ to *KEY*. If such a list is not found, ASSOC returns NIL. Example:

```
(ASSOC 'B '((A . 1) (B . 2) (C . 3))) => (B . 2)
```

(**FASSOC** *KEY ALST*)   [Function]

Faster version of ASSOC that terminates on a null-check.

(**SASSOC** *KEY ALST*)   [Function]

Same as ASSOC, but uses EQUAL instead of EQ when searching for *KEY*.

(**PUTASSOC** *KEY VAL ALST*)   [Function]

Searches *ALST* for a sublist CAR of which is EQ to *KEY*. If one is found, the CDR is replaced
(using RPLACD) with *VAL*. If no such sublist is found, (CONS *KEY VAL*) is added at the
end of *ALST*. Returns *VAL*. If *ALST* is not a list, generates an error, Arg not list.

The argument order for ASSOC, PUTASSOC, etc. is different from that of LISTGET, LISTPUT, etc.

(**LISTGET** *LST PROP*)   [Function]

Searches *LST* two elements at a time, by CDDR, looking for an element EQ to *PROP*. If one
is found, returns the next element of *LST*, otherwise NIL. Returns NIL if *LST* is not a list.
Example:

```
(LISTGET '(A 1 B 2 C 3) 'B) => 2
(LISTGET '(A 1 B 2 C 3) 'W) => NIL
```

(**LISTPUT** *LST PROP VAL*)   [Function]

Searches *LST* two elements at a time, by CDDR, looking for an element EQ to *PROP*. If
PROP is found, replaces the next element of *LST* with *VAL*. Otherwise, *PROP* and *VAL* are
added to the end of *LST*. If *LST* is a list with an odd number of elements, or ends in a
non-list other than NIL, *PROP* and *VAL* are added at its beginning. Returns *VAL*. If *LST* is
not a list, generates an error, Arg not list.

(**LISTGET1** *LST PROP*)   [Function]

Like LISTGET, but searches *LST* one CDR at a time, i.e., looks at each element. Returns the
next element after *PROP*. Examples:

```
(LISTGET1 '(A 1 B 2 C 3) 'B) => 2
(LISTGET1 '(A 1 B 2 C 3) '1) => B
(LISTGET1 '(A 1 B 2 C 3) 'W) => NIL
```

(**LISTPUT1** *LST PROP VAL*)   [Function]

Like LISTPUT, but searches *LST* one CDR at a time. Returns the modified *LST*. Example:

```
←(SETQ FOO '(A 1 B 2))
   (A 1 B 2)
←(LISTPUT1 FOO 'B 3)
   (A 1 B 3)
←(LISTPUT1 FOO 'C 4)
   (A 1 B 3 C 4)
←(LISTPUT1 FOO 1 'W)
   (A 1 W 3 C 4)
←FOO
```

```
                            (A 1 W 3 C 4)
```

If *LST* is not a list, no error is generated. However, since a non-list cannot be changed into a list, *LST* is not modified. In this case, the value of LISTPUT1 should be saved. Example:

```
←(SETQ FOO NIL)
   NIL
←(LISTPUT1 FOO 'A 5)
   (A 5)
←FOO
   NIL
```

Sorting Lists

(**SORT** *DATA COMPAREFN*)   [Function]

DATA is a list of items to be sorted using *COMPAREFN*, a predicate function of two arguments which can compare any two items on *DATA* and return T if the first one belongs before the second. If *COMPAREFN* is NIL, ALPHORDER is used; thus (SORT *DATA*) will alphabetize a list. If *COMPAREFN* is T, CAR's of items that are lists are given to ALPHORDER, otherwise the items themselves; thus (SORT A-LIST T) will alphabetize an assoc list by the CAR of each item. (SORT X 'ILESSP) will sort a list of integers.

The value of SORT is the sorted list. The sort is destructive and uses no extra storage. The value returned is EQ to *DATA* but elements have been switched around. There is no safe way to interrupt SORT. If you abort a call to SORT by any means, you may loose elements from the list beeing sorted. The algorithm used by SORT is such that the maximum number of compares is $N*\log_2 N$, where *N* is (LENGTH *DATA*).

> Note: If (*COMPAREFN* A B) = *(COMPAREFN* B A), then the ordering of A and B may or may not be preserved.

For example, if (FOO . FIE) appears before (FOO . FUM) in X, (SORT X T) may or may not reverse the order of these two elements.

(**MERGE** *A B COMPAREFN*)   [Function]

A and *B* are lists which have previously been sorted using SORT and *COMPAREFN*. Value is a destructive merging of the two lists. It does not matter which list is longer. After merging both *A* and *B* are equal to the merged list. (In fact, (CDR *A*) is EQ to (CDR *B*)).

(**ALPHORDER** *A B CASEARRAY*)   [Function]

A predicate function of two arguments, for alphabetizing. Returns a non-NIL value if its arguments are in lexicographic order, i.e., if *B* does not belong before *A*. Numbers come before literal atoms, and are ordered by magnitude (using GREATERP). Literal atoms and strings are ordered by comparing the character codes in their print names. Thus (ALPHORDER 23 123) is T, whereas (ALPHORDER 'A23 'A123) is NIL, because the character code for the digit 2 is greater than the code for 1.

Atoms and strings are ordered before all other data types. If neither *A* nor *B* are atoms or strings, the value of ALPHORDER is always T.

If *CASEARRAY* is non-NIL, it is a casearray (see the Random Access File Operations section of Chapter 25) that the characters of *A* and *B* are translated through before being compared. Numbers are not passed through *CASEARRAY*.

Note: If either *A* or *B* is a number, the value returned in the "true" case is T. Otherwise, ALPHORDER returns either EQUAL or LESSP to discriminate the cases of *A* and *B* being equal or unequal strings/atoms.

Note: ALPHORDER does no UNPACKs, CHCONs, CONSes or NTHCHARs. It is several times faster for alphabetizing than anything that can be written using these other functions.

(**UALPHORDER** *A B*)   [Function]

Defined as (ALPHORDER *A B* UPPERCASEARRAY). UPPERCASEARRAY maps every lowercase character into the corresponding uppercase character. For more information on UPPERCASEARRAY see Chapter 25.

(**MERGEINSERT** *NEW LST ONEFLG*)   [Function]

LST is NIL or a list of partially sorted items. MERGEINSERT tries to find the "best" place to (destructively) insert *NEW*, e.g.,

```
(MERGEINSERT 'FIE2 '(FOO FOO1 FIE FUM)) => (FOO FOO1 FIE
    FIE2 FUM)
```

Returns *LST*. MERGEINSERT is undoable.

If *ONEFLG* = T and *NEW* is already a member of *LST*, MERGEINSERT does nothing and returns *LST*.

MERGEINSERT is used by ADDTOFILE (see the Functions for Manipulating File Command Lists section of Chapter 17) to insert the name of a new function into a list of functions. The algorithm is essentially to look for the item with the longest common leading sequence of characters with respect to *NEW*, and then merge *NEW* in starting at that point.

Other List Functions

(**REMOVE** *X L*)   [Function]

Removes all top-level occurrences of *X* from list *L*, returning a copy of *L* with all elements EQUAL to *X* removed. Example:

```
(REMOVE 'A '(A B C (A) A))  =>  (B C (A))
(REMOVE '(A) '(A B C (A) A))  =>  (A B C A)
```

(**DREMOVE** *X L*)   [Function]

Like REMOVE, but uses EQ instead of EQUAL, and actually modifies the list *L* when removing *X*, and thus does not use any additional storage. More efficient than REMOVE.

DREMOVE cannot *change* a list to NIL:

```
←(SETQ FOO '(A))
```

```
            (A)
    ←(DREMOVE 'A FOO)
        NIL
    ←FOO
        (A)
```

The DREMOVE above returns NIL, and does not perform any CONSes, but the value of FOO is *still* (A), because there is no way to change a list to a non-list. See NCONC.

(**REVERSE** *L*)   [Function]

Reverses (and copies) the top level of a list, e.g.,

```
    (REVERSE '(A B (C D)))  =>  ((C D) B A)
```

If *L* is not a list, REVERSE just returns *L*.

(**DREVERSE** *L*)   [Function]

Value is the same as that of REVERSE, but DREVERSE destroys the original list *L* and thus does not use any additional storage. More efficient than REVERSE.

(**COMPARELISTS** *X Y*)   [Function]

Compares the list structures *X* and *Y* and prints a description of any differences to the terminal. If *X* and *Y* are EQUAL lists, COMPARELISTS simply prints out SAME. Returns NIL.

COMPARELISTS prints a terse description of the differences between the two list structures, highlighting the items that have changed. This printout is not a complete and perfect comparison. If *X* and *Y* are radically different list structures, the printout will not be very useful. COMPARELISTS is meant to be used as a tool to help users isolate differences between similar structures.

When a single element has been changed for another, COMPARELISTS prints out items such as (*A* -> *B*), for example:

```
    ←(COMPARELISTS '(A B C D) '(X B E D))
        (A -> X) (C -> E)
        NIL
```

When there are more complex differences between the two lists, COMPARELISTS prints *X* and *Y*, highlighting differences and abbreviating similar elements as much as possible. "&" is used to signal a single element that is present in the same place in the two lists; "--" signals an arbitrary number of elements in one list but not in the other; "-2-," "-3-", etc. signal a sequence of two, three, etc. elements that are the same in both lists. Examples:

```
    (COMPARELISTS '(A B C D) '(A D))
        (A B C --)
        (A D)
    ←(COMPARELISTS '(A B C D E F G H) '(A B C D X))
        (A -3- E F --)
        (A -3- X)
    ←(COMPARELISTS '(A B C (D E F (G) H) I) '(A B (G) C (D E F
      H) I))
        (A &      & (D -2- (G) &) &)
```

```
(A & (G) & (D -2-     &) &)
```

(**NEGATE** *X*)   [Function]

    For a form *X*, returns a form which computes the negation of *X*. For example:

```
(NEGATE '(MEMBER X Y)) => (NOT (MEMBER X Y))
(NEGATE '(EQ X Y)) => (NEQ X Y)
(NEGATE '(AND X (NLISTP X))) => (OR (NULL X) (LISTP X))
(NEGATE NIL) => T
```

[This page intentionally left blank]

A string represents a sequence of characters. Interlisp strings are a subtype of Common Lisp strings. Medley provides functions for creating strings, concatenating strings, and creating sub-strings of a string; all accepting or producing Common Lisp-acceptable strings.

A string is typed as a double quote (`"`), followed by a sequence of any characters except double quote and `%`, terminated by a double quote. To include `%` or `"` in a string, type `%` in front of them:

```
"A string"
"A string with %" in it, and a %%."
""        ; an empty string
```

Strings are printed by `PRINT` and `PRIN2` with initial and final double quotes, and `%`s inserted where necessary for it to read back in properly. Strings are printed by `PRIN1` without the double quotes and extra `%`s. The null string is printed by `PRINT` and `PRIN2` as `""`. (`PRIN1 ""`) doesn't print anything.

Internally, a string is stored in two parts: a "string header" and the sequence of characters. Several string headers may refer to the the same character sequence, so a substring can be made by creating a new string header, without copying any characters. Functions that refer to "strings" actually manipulate string headers. Some functions take an "old string" argument, and re-use the string pointer.

(**STRINGP** *X*) [Function]

Returns *X* if *X* is a string, `NIL` otherwise.

(**STREQUAL** *X Y*) [Function]

Returns `T` if *X* and *Y* are both strings and they contain the same sequence of characters, otherwise `NIL`. `EQUAL` uses `STREQUAL`. Note that strings may be `STREQUAL` without being `EQ`. For instance,

```
(STREQUAL "ABC" "ABC")  =>  T
(EQ "ABC" "ABC")  =>  NIL
```

`STREQUAL` returns `T` if *X* and *Y* are the same string pointer, or two different string pointers which point to the same character sequence, or two string pointers which point to different character sequences which contain the same characters. Only in the first case would *X* and *Y* be `EQ`.

(**STRING-EQUAL** *X Y*) [Function]

Returns `T` if *X* and *Y* are either strings or symbols, and they contain the same sequence of characters, ignoring case. For instance,

```
(STRING-EQUAL "FOO" "Foo")  =>  T
(STRING-EQUAL "FOO" 'Foo)  =>  T
```

This is useful for comparing things that might want to be considered "equal" even though they're not both symbols in a consistent case, such as file names and user names.

(**STRING.EQUAL** *X Y*) [Function]

Returns T if the print names of *X* and *Y* contain the same sequence of characters, ignoring case. For instance,

```
(STRING-EQUAL "320" 320)  =>  T
(STRING-EQUAL "FOO" 'Foo)  =>  T
```

This is like STRING-EQUAL, but handles numbers, etc., where STRING-EQUAL doesn't.

(**ALLOCSTRING** *N INITCHAR OLD FATFLG*) [Function]

Creates a string of length *N* characters of *INITCHAR* (which can be either a character code or something coercible to a character). If *INITCHAR* is NIL, it defaults to character code 0. if *OLD* is supplied, it must be a string pointer, which is modified and returned.

If *FATFLG* is non-NIL, the string is allocated using full 16-bit NS characters (see Chapter 2) instead of 8-bit characters. This can speed up some string operations if NS characters are later inserted into the string. This has no other effect on the operation of the string functions.

(**MKSTRING** *X FLG RDTBL*) [Function]

If *X* is a string, returns *X*. Otherwise, creates and returns a string containing the print name of *X*. Examples:

```
(MKSTRING "ABC")  =>  "ABC"
(MKSTRING '(A B C))  =>  "(A B C)"
(MKSTRING NIL)  =>  "NIL"
```

Note that the last example returns the string "NIL", not the symbol NIL.

If *FLG* is T, then the PRIN2-name of *X* is used, computed with respect to the readtable *RDTBL*. For example,

```
(MKSTRING "ABC" T)  =>  "%"ABC%""
```

(**NCHARS** *X FLG RDTBL*) [Function]

Returns the number of characters in the print name of *X*. If *FLG*=T, the PRIN2-name is used. For example,

```
(NCHARS 'ABC)  =>  3
(NCHARS "ABC" T)  =>  5
```

Note: NCHARS works most efficiently on symbols and strings, but can be given any object.

(**SUBSTRING** *X N M OLDPTR*) [Function]

Returns the substring of *X* consisting of the *N*th through *M*th characters of *X*. If *M* is NIL, the substring contains the *N*th character thru the end of *X*. *N* and *M* can be negative numbers, which are interpreted as counts back from the end of the string, as with NTHCHAR (Chapter 2). SUBSTRING returns NIL if the substring is not well defined, (e.g., *N* or *M* specify character positions outside of *X*, or *N* corresponds to a character in *X* to the right of the character indicated by *M*). Examples:

```
(SUBSTRING "ABCDEFG" 4 6)  =>  "DEF"
(SUBSTRING "ABCDEFG" 3 3)  =>  "C"
(SUBSTRING "ABCDEFG" 3 NIL)  =>  "CDEFG"
(SUBSTRING "ABCDEFG" 4 -2)  =>  "DEF"
(SUBSTRING "ABCDEFG" 6 4)  =>  NIL
(SUBSTRING "ABCDEFG" 4 9)  =>  NIL
```

If *X* is not a string, it is converted to one. For example,

```
(SUBSTRING '(A B C) 4 6)  =>  "B C"
```

SUBSTRING does not actually copy any characters, but simply creates a new string pointer to the characters in *X*. If *OLDPTR* is a string pointer, it is modified and returned.

(**GNC** *X*)   [Function]

"Get Next Character." Returns the next character of the string *X* (as a symbol); also removes the character from the string, by changing the string pointer. Returns NIL if *X* is the null string. If *X* isn't a string, a string is made. Used for sequential access to characters of a string. Example:

```
←(SETQ FOO "ABCDEFG")
    "ABCDEFG"
←(GNC FOO)
    A
←(GNC FOO)
    B
←FOO
    "CDEFG"
```

Note that if A is a substring of B, (GNC A) does not remove the character from B.

(**GLC** *X*)   [Function]

"Get Last Character." Returns the last character of the string *X* (as a symbol); also removes the character from the string. Similar to GNC. Example:

```
←(SETQ FOO "ABCDEFG")
    "ABCDEFG"
←(GLC FOO)
    G
←(GLC FOO)
    F
←FOO
    "ABCDE"
```

(**CONCAT** *X₁ X₂ ... Xₙ*)   [NoSpread Function]

Returns a new string which is the concatenation of (copies of) its arguments. Any arguments which are not strings are transformed to strings. Examples:

```
(CONCAT "ABC" 'DEF "GHI")  =>  "ABCDEFGHI"
(CONCAT '(A B C) "ABC")  =>  "(A B C)ABC"
(CONCAT) returns the null string,  ""
```

(**CONCATLIST** *L*)   [Function]

    L is a list of strings and/or other objects. The objects are transformed to strings if they aren't strings. Returns a new string which is the concatenation of the strings. Example:

        (CONCATLIST '(A B (C D) "EF")) => "AB(C D)EF"

(**RPLSTRING** *X N Y*)   [Function]

    Replaces the characters of string *X* beginning at character position *N* with string *Y*. *X* and *Y* are converted to strings if they aren't already. *N* may be positive or negative, as with SUBSTRING. Characters are smashed into (converted) *X*. Returns the string *X*. Examples:

        (RPLSTRING "ABCDEF" –3 "END") => "ABCEND"
        (RPLSTRING "ABCDEFGHIJK" 4 '(A B C)) => "ABC(A B C)K"

    Generates an error if there is not enough room in *X* for *Y*, i.e., the new string would be longer than the original. If *Y* was not a string, *X* will already have been modified since RPLSTRING does not know whether *Y* will "fit" without actually attempting the transfer.

    Warning: In some implementations of Interlisp, if *X* is a substring of *Z*, *Z* will also be modified by the action of RPLSTRING or RPLCHARCODE. However, this is not guaranteed to be true in all cases, so programmers should not rely on RPLSTRING or RPLCHARCODE altering the characters of any string other than the one directly passed as argument to those functions.

(**RPLCHARCODE** *X N CHAR*)   [Function]

    Replaces the *N*th character of the string *X* with the character code *CHAR*. *N* may be positive or negative. Returns the new *X*. Similar to RPLSTRING. Example:

        (RPLCHARCODE "ABCDE" 3 (CHARCODE F)) => "ABFDE"

(**STRPOS** *PAT STRING START SKIP ANCHOR TAIL CASEARRAY BACKWARDSFLG*)   [Function]

    STRPOS is a function for searching one string looking for another. *PAT* and *STRING* are both strings (or else they are converted automatically). STRPOS searches *STRING* beginning at character number *START*, (or 1 if *START* is NIL) and looks for a sequence of characters equal to *PAT*. If a match is found, the character position of the first matching character in *STRING* is returned, otherwise NIL. Examples:

        (STRPOS "ABC" "XYZABCDEF") => 4
        (STRPOS "ABC" "XYZABCDEF" 5) => NIL
        (STRPOS "ABC" "XYZABCDEFABC" 5) => 10

    SKIP can be used to specify a character in *PAT* that matches any character in *STRING*. Examples:

        (STRPOS "A&C&" "XYZABCDEF" NIL '&) => 4
        (STRPOS "DEF&" "XYZABCDEF" NIL '&) => NIL

    If *ANCHOR* is T, STRPOS compares *PAT* with the characters beginning at position *START* (or 1 if *START* is NIL). If that comparison fails, STRPOS returns NIL without searching any further down *STRING*. Thus it can be used to compare one string with some *portion* of another string. Examples:

        (STRPOS "ABC" "XYZABCDEF" NIL NIL T) => NIL

```
(STRPOS "ABC" "XYZABCDEF" 4 NIL T)  =>  4
```

If *TAIL* is T, the value returned by STRPOS if successful is not the starting position of the sequence of characters corresponding to *PAT*, but the position of the first character after that, i.e., the starting position plus (NCHARS *PAT*). Examples:

```
(STRPOS "ABC" "XYZABCDEFABC" NIL NIL NIL T)  =>  7
(STRPOS "A" "A" NIL NIL NIL T)  =>  2
```

If *TAIL* = NIL, STRPOS returns NIL, or a character position within *STRING* which can be passed to SUBSTRING. In particular, (STRPOS "" "") => NIL. However, if *TAIL* = T, STRPOS may return a character position outside of *STRING*. For instance, note that the second example above returns 2, even though "A" has only one character.

If *CASEARRAY* is non-NIL, this should be a casearray like that given to FILEPOS (Chapter 25). The casearray is used to map the string characters before comparing them to the search string.

If *BACKWARDSFLG* is non-NIL, the search is done backwards from the end of the string.

(**STRPOSL** *A STRING START NEG BACKWARDSFLG*)   [Function]

STRING is a string (or is converted automatically to a string), *A* is a list of characters or character codes. STRPOSL searches *STRING* beginning at character number *START* (or 1 if *START* = NIL) for one of the characters in *A*. If one is found, STRPOSL returns as its value the corresponding character position, otherwise NIL. Example:

```
(STRPOSL '(A B C) "XYZBCD")  =>  4
```

If *NEG* = T, STRPOSL searches for a character *not* on *A*. Example:

```
(STRPOSL '(A B C) "ABCDEF" NIL T)  =>  4
```

If any element of *A* is a number, it is assumed to be a character code. Otherwise, it is converted to a character code via CHCON1. Therefore, it is more efficient to call STRPOSL with *A* a list of character *codes.*

If *A* is a bit table, it is used to specify the characters (see MAKEBITTABLE below)

If *BACKWARDSFLG* is non-NIL, the search is done backwards from the end of the string.

STRPOSL uses a "bit table" data structure to search efficiently. If *A* is not a bit table, it is converted to a bit table using MAKEBITTABLE. If STRPOSL is to be called frequently with the same list of characters, a considerable savings can be achieved by converting the list to a bit table *once*, and then passing the bit table to STRPOSL as its first argument.

(**MAKEBITTABLE** *L NEG A*)   [Function]

Returns a bit table suitable for use by STRPOSL. *L* is a list of characters or character codes, *NEG* is the same as described for STRPOSL. If *A* is a bit table, MAKEBITTABLE modifies and returns it. Otherwise, it will create a new bit table.

Note: If *NEG* = T, STRPOSL must call MAKEBITTABLE whether *A* is a list *or* a bit table. To obtain bit table efficiency with *NEG*=T, MAKEBITTABLE should be called with *NEG*=T, and the resulting "inverted" bit table should be given to STRPOSL with *NEG*=NIL.

[This page intentionally left blank]

An Interlisp array is a one-dimensional vector of objects. Arrays are generally created by the function ARRAY. By contrast, Common Lisp arrays can be multi-dimensional.

Note: Interlisp arrays and Common Lisp arrays are *not* the same types. Interlisp functions only accept Interlisp arrays and vice versa. There are no functions to convert between the two types.

(**ARRAY** *SIZE TYPE INIT ORIG* −)   [Function]

> Creates and returns a new array that holds *SIZE* objects of type *TYPE*. If *TYPE* is NIL, the array can contain any arbitrary Lisp datum. In general, *TYPE* may be any of the various field specifications that are legal in DATATYPE declarations (see Chapter 8): POINTER, FIXP, FLOATP, (BITS *N*), etc. Medley will, if necessary, choose an "enclosing" type if the given one is not supported; for example, an array of (BITS 3) may be represented by an array of (BITS 8).

> *INIT* is the initial value for each element of the new array. If not specified, the array elements will be initialized with 0 (for number arrays) or NIL (all other types).

> Arrays can have either 0-origin or 1-origin indexing, as specified by the *ORIG* argument; if *ORIG* is not specified, the default is 1.

> Arrays of type FLOATP are stored unboxed. This increases the space and time efficiency of FLOATP arrays. If you want to use boxed floating point numbers, use an array of type POINTER instead of FLOATP.

(**ARRAYP** *X*)   [Function]

> Returns X if *X* is an array, NIL otherwise.

(**ELT** *ARRAY N*)   [Function]

> Returns the *N*th element of the array *ARRAY*.

> Causes the error, Arg not array, if *ARRAY* is not an array. Causes the error, Illegal Arg, if *N* is out of bounds.

(**SETA** *ARRAY N VAL*)   [Function]

> Sets the *N*th element of *ARRAY* to *VAL*, and returns *VAL*.

> Causes the error, Arg not array, if *ARRAY* is not an array. the error, Illegal Arg, if *N* is out of bounds. Can cause the error, Non-numeric arg, if *ARRAY* is an array whose ARRAYTYP is FIXP or FLOATP and *VAL* is non-numeric.

(**ARRAYTYP** *ARRAY*)   [Function]

> Returns the type of the elements in *ARRAY*, a value corresponding to the second argument to ARRAY.

If ARRAY coerced the array type as described above, ARRAYTYP returns the *new* type. For example, (ARRAYTYP (ARRAY 10 '(BITS 3))) returns BYTE.

(**ARRAYSIZE** *ARRAY*)   [Function]

Returns the size of *ARRAY*. Generates the error, Arg not array, if *ARRAY* is not an array.

(**ARRAYORIG** *ARRAY*)   [Function]

Returns the origin of *ARRAY*, which may be 0 or 1. Generates an error, Arg not array, if *ARRAY* is not an array.

(**COPYARRAY** *ARRAY*)   [Function]

Returns a new array of the same size and type as *ARRAY*, and with the same contents as *ARRAY*. Generates an error, Arg not array, if *ARRAY* is not an array.

[This page intentionally left blank]

Hash arrays let you associate arbitrary Lisp objects ("hash keys") with other objects ("hash values"), so you can get from key to value quickly. There are functions for creating hash arrays, putting a hash key/value pair in a hash array, and quickly retrieving the hash value associated with a given hash key.

By default, the hash array functions use EQ for comparing hash keys. This means that if non-symbols are used as hash keys, the exact same object (not a copy) must be used to retrieve the hash value. However, you can specify the function used to compare hash keys and to "hash" a hash key to a number. You can, for example, create hash arrays where EQUAL but non-EQ strings will hash to the same value. Specifying alternative hashing algorithms is described below.

In the description of the functions below, the argument *HARRAY* should be a hasharray created by HASHARRAY. For convenience in interactive program development, it may also be NIL, in which case a hash array (SYSHASHARRAY) provided by the system is used; you must watch out for confusions if this form is used to associate more than one kind of value with the same key.

Note: For backwards compatibility, the hash array functions will accept a list whose CAR is a hash array, and whose CDR is the "overflow method" for the hash array (see below). However, hash array functions are guaranteed to perform with maximum efficiency only if a direct value of HASHARRAY is given.

Note: Interlisp hash arrays and Common Lisp hash tables are the same data type, so functions from both may be intermixed. The only difference between the functions may be argument order, as in MAPHASH and CL:MAPHASH (see below).

> (**HASHARRAY** *MINKEYS OVERFLOW HASHBITSFN EQUIVFN RECLAIMABLE REHASH-*
> *THRESHOLD*) [Function]
>
> > Creates a hash array with space for at least *MINKEYS* hash keys, with overflow method *OVERFLOW*. See discussion of overflow behavior below.
> >
> > If *HASHBITSFN* and *EQUIVFN* are non-NIL, they specify the hashing function and comparison function used to interpret hash keys. This is described in the section on user-specified hashing functions below. If *HASHBITSFN* and *EQUIVFN* are NIL, the default is to hash EQ hash keys to the same value.
> >
> > If *RECLAIMABLE* is *T* the entries in the hash table will be removed if the key has a reference count of one and the table is about to be rehashed. This allows the system, in some cases, to reuse keys instead of expanding the table.
> >
> > **Note**: CL:MAKE-HASH-TABLE does not allow you to specify your own hashing functions but does provide three built-in types specified by *Common Lisp, the Language*.

> (**HARRAY** *MINKEYS*) [Function]
>
> > Provided for backward compatibility, this is equivalent to (HASHARRAY MINKEYS 'ERROR), i.e. if the resulting hasarray gets full, an error occurs.

(**HARRAYP** *X*)   [Function]

> Returns *X* if it is a hash array; otherwise NIL.

> HARRAYP returns NIL if *X* is a list whose CAR is an HARRAYP, even though this is accepted by the hash array functions (see below).

(**PUTHASH** *KEY VAL HARRAY*)   [Function]

> Associates the hash value *VAL* with the hash key *KEY* in *HARRAY*. Replaces the previous hash value, if any. If *VAL* is NIL, any old association is removed (hence a hash value of NIL is not allowed). If *HARRAY* is full when PUTHASH is called with a key not already in the hash array, the function HASHOVERFLOW is called, and the PUTHASH is applied to the value returned (see below). Returns *VAL*.

(**GETHASH** *KEY HARRAY*)   [Function]

> Returns the hash value associated with the hash key *KEY* in *HARRAY*. Returns NIL, if *KEY* is not found.

(**CLRHASH** *HARRAY*)   [Function]

> Clears all hash keys/values from *HARRAY*. Returns *HARRAY*.

(**HARRAYPROP** *HARRAY PROP NEWVALUE*)   [NoSpread Function]

> Returns the property *PROP* of *HARRAY*; *PROP* can have the system-defined values SIZE (the maximum occupancy of *HARRAY*), NUMKEYS (number of occupied slots), OVERFLOW (overflow method), HASHBITSFN (hashing function) and EQUIVFN (comparison function). Except for SIZE and NUMKEYS, a new value may be specified as *NEWVALUE*.

> By using other values for *PROP*, the user may also set and get arbitrary property values, to associate additional information with a hash array.

> The HASHBITSFN or EQUIVFN properties can only be changed if the hash array is empty.

(**HARRAYSIZE** *HARRAY*)   [Function]

> Returns the number of slots in *HARRAY*. It's equivalent to (HARRAYPROP HARRAY 'SIZE).

(**REHASH** *OLDHARRAY NEWHARRAY*)   [Function]

> Hashes all hash keys and values in *OLDHARRAY* into *NEWHARRAY*. The two hash arrays do not have to be (and usually aren't) the same size. Returns *NEWHARRAY*.

(**MAPHASH** *HARRAY MAPHFN*)   [Function]

> *MAPHFN* is a function of two arguments. For each hash key in *HARRAY*, *MAPHFN* will be applied to the hash value, and the hash key. For example:

```
[MAPHASH A
    (FUNCTION (LAMBDA (VAL KEY)
        (if (LISTP KEY) then (PRINT VAL)]
```

> will print the hash value for all hash keys that are lists. MAPHASH returns *HARRAY*.

Note: the argument order for CL:MAPHASH is *MAPHFN HARRAY*.

(**DMPHASH** *HARRAY₁ HARRAY₂ ... HARRAYₙ*)   [NLambda NoSpread Function]

Prints on the primary output file LOADable forms which will restore the hash-arrays contained as the values of the atoms *HARRAY₁, HARRAY₂, ... HARRAYₙ*. Example: (DMPHASH SYSHASHARRAY) will dump the system hash-array.

All EQ identities except symbols and small integers are lost by dumping and loading because READ will create new structure for each item. Thus if two lists contain an EQ substructure, when they are dumped and loaded back in, the corresponding substructures while EQUAL are no longer EQ. The HORRIBLEVARS file package command (Chapter 17) provides a way of dumping hash tables such that these identities are preserved.

Hash Overflow

When a hash array becomes full, trying to add another hash key will cause the function HASHOVERFLOW to be called. This either enlarges the hash array, or causes the error Hash table full. How hash overflow is handled is determined by the value of the OVERFLOW property of the hash array (which can be accessed by HARRAYPROP). The possibilities for the overflow method are:

the symbol ERROR	The error Hash array full is generated when the hash array overflows. This is the default overflow behavior for hash arrays returned by HARRAY.
NIL	The array is automatically enlarged by at least a factor 1.5 every time it overflows. This is the default overflow behavior for hash arrays returned by HASHARRAY.
a positive integer *N*	The array is enlarged to include at least *N* more slots than it currently has.
a floating point number *F*	The array is changed to include *F* times the number of current slots.
a function or lambda expression *FN*	Upon hash overflow, *FN* is called with the hash array as its argument. If *FN* returns a number, that will become the size of the array. Otherwise, the new size defaults to 1.5 times its previous size. *FN* could be used to print a message, or perform some monitor function.

Note: For backwards compatibility, the hash array functions accept a list whose CAR is the hash array, and whose CDR is the overflow method. In this case, the overflow method specified in the list overrides the overflow method set in the hash array. Hash array functions perform with maximum efficiency only if a direct value of HASHARRAY is given.

Specifying Your Own Hashing Functions

In general terms, when a key is looked up in a hash array, it is converted to an integer, which is used to index into a linear array. If the key is not the same as the one found at that index, other indices are

tried until it the desired key is found. The value stored with that key is then returned (from GETHASH) or replaced (from PUTHASH).

To customize hash arrays, you'll need to supply the "hashing function" used to convert a key to an integer and the comparison function used to compare the key found in the array with the key being looked up. For hash arrays to work correctly, any two objects which are equal according to the comparison function must "hash" to equal integers.

By default, Medley uses a hashing function that computes an integer from the internal address of a key, and use EQ for comparing keys. This means that if non-atoms are used as hash keys, *the exact same object* (not a copy) must be used to retrieve the hash value.

There are some applications for which the EQ constraint is too restrictive. For example, it may be useful to use strings as hash keys, without the restriction that EQUAL but not EQ strings are considered to be different hash keys.

The user can override this default behavior for any hash array by specifying the functions used to compare keys and to "hash" a key to a number. This can be done by giving the *HASHBITSFN* and *EQUIVFN* arguments to HASHARRAY (see above).

The *EQUIVFN* argument is a function of two arguments that returns non-NIL when its arguments are considered equal. The *HASHBITSFN* argument is a function of one argument that produces a positive small integer (in the range $[0..2^{16} - 1]$) with the property that objects that are considered equal by the *EQUIVFN* produce the same hash bits.

For an existing hash array, the function HARRAYPROP (see above) can be used to examine the hashing and equivalence functions as the HASHBITSFN and EQUIVFN hash array properties. These properties are read-only for non-empty hash arrays, as it makes no sense to change the equivalence relationship once some keys have been hashed.

The following function is useful for creating hash arrays that take strings as hash keys:

(**STRINGHASHBITS** *STRING*)   [Function]

> Hashes the string *STRING* into an integer that can be used as a HASHBITSFN for a hash array. Strings which are STREQUAL hash to the same integer.
>
> Example:
>
>> (HASHARRAY *MINKEYS OVERFLOW* 'STRINGHASHBITS 'STREQUAL)
>
> creates a hash array where you can use strings as hash keys.

INTERLISP-D REFERENCE MANUAL

[This page intentionally left blank]

There are four different types of numbers in Interlisp: small integers, large integers, bignums (arbitrary-size integers), and floating-point numbers. Small integers are in the range -65536 to 65535. Large integers and floating-point numbers are 32-bit quantities that are stored by "boxing" the number (see below). Bignums are "boxed" as a series of words.

Large integers and floating-point numbers can be any full word quantity. To distinguish among the various kinds of numbers, and other Interlisp pointers, these numbers are "boxed" When a large integer or floating-point number is created (by an arithmetic operation or by READ), Interlisp gets a new word from "number storage" and puts the number into that word. Interlisp then passes around the pointer to that word, i.e., the "boxed number", rather than the actual quantity itself. When a numeric function needs the actual numeric quantity, it performs the extra level of addressing to obtain the "value" of the number. This latter process is called "unboxing". Unboxing does not use any storage, but each boxing operation uses one new word of number storage. If a computation creates many large integers or floating-point numbers, i.e., does lots of boxes, it may cause a garbage collection of large integer space, or of floating-point number space.

The following functions can be used to distinguish the different types of numbers:

(**SMALLP** *X*)   [Function]

> Returns *X*, if *X* is a small integer; NIL otherwise. Does not generate an error if *X* is not a number.

(**FIXP** *X*)   [Function]

> Returns *X*, if *X* is an integer; NIL otherwise. Note that FIXP is true for small integers, large integers, and bignums. Does not generate an error if *X* is not a number.

(**FLOATP** *X*)   [Function]

> Returns *X* if *X* is a floating-point number; NIL otherwise. Does not give an error if *X* is not a number.

(**NUMBERP** *X*)   [Function]

> Returns *X*, if *X* is a number of any type; NIL otherwise. Does not generate an error if *X* is not a number.

> **Note**: In previous releases, NUMBERP was true only if (FLOATP *X*) or (FIXP *X*) were true. With the additon of Common Lisp ratios and complex numbers, NUMBERP now returns T for *all* number types . Code relying on the "old" behavior should be modified.

Each small integer has a unique representation, so EQ may be used to check equality. EQ should not be used for large integers, bignums, or floating-point numbers, EQP, IEQP, or EQUAL must be used instead.

(**EQP** *X Y*)   [Function]

> Returns T, if *X* and *Y* are equal numbers; NIL otherwise. EQ may be used if *X* and *Y* are known to be small integers. EQP does not convert *X* and *Y* to integers, e.g., (EQP 2000

2000.3) => NIL, but it can be used to compare an integer and a floating-point number, e.g., (EQP 2000 2000.0) => T. EQP does not generate an error if *X* or *Y* are not numbers.

EQP can also be used to compare stack pointers (see Chapter 11) and compiled code objects (see Chapter 10).

The action taken on division by zero and floating-point overflow is determined with the following function:

(**OVERFLOW** *FLG*)   [Function]

Sets a flag that determines the system response to arithmetic overflow (for floating-point arithmetic) and division by zero; returns the previous setting.

For integer arithmetic: If *FLG* = T, an error occurs on division by zero. If *FLG* = NIL or 0, integer division by zero returns zero. Integer overflow cannot occur, because small integers are converted to bignums (see the beginning of this chapter).

For floating-point arithmetic: If *FLG* = T, an error occurs on floating overflow or floating division by zero. If *FLG* = NIL or 0, the largest (or smallest) floating-point number is returned as the result of the overflowed computation or floating division by zero.

The default value for OVERFLOW is T, meaning an error is generated on division by zero or floating overflow.

Generic Arithmetic

The functions in this section are "generic" arithmetic functions. If any of the arguments are floating-point numbers (see the Floating-Point Arithmetic section below), they act exactly like floating-point functions, floating all arguments and returning a floating-point number as their value. Otherwise, they act like the integer functions (see the Integer Arithmetic section below). If given a non-numeric argument, they generate an error, Non-numeric arg. The results of division by zero and floating-point overflow is determined by the function OVERFLOW (see the section above).

(**PLUS** X_1 X_2 ... X_N)   [NoSpread Function]

$X_1 + X_2 + ... + X_N$.

(**MINUS** *X*)   [Function]

$- X$

(**DIFFERENCE** *X Y*)   [Function]

$X - Y$

(**TIMES** X_1 X_2 ... X_N)   [NoSpread Function]

$X_1 * X_2 * ... * X_N$

(**QUOTIENT** *X Y*) [Function]

> If *X* and *Y* are both integers, returns the integer division of *X* and *Y*. Otherwise, converts both *X* and *Y* to floating-point numbers, and does a floating-point division.

(**REMAINDER** *X Y*) [Function]

> If *X* and *Y* are both integers, returns (IREMAINDER *X Y*), otherwise (FREMAINDER *X Y*).

(**GREATERP** *X Y*) [Function]

> T, if *X* > *Y*, NIL otherwise.

(**LESSP** *X Y*) [Function]

> T if *X* < *Y*, NIL otherwise.

(**GEQ** *X Y*) [Function]

> T, if *X* >= *Y*, NIL otherwise.

(**LEQ** *X Y*) [Function]

> T, if *X* <= *Y*, NIL otherwise.

(**ZEROP** *X*) [Function]

> The same as (EQP *X* 0).

(**MINUSP** *X*) [Function]

> T, if *X* is negative; NIL otherwise. Works for both integers and floating-point numbers.

(**MIN** X_1 X_2 . . . X_N) [NoSpread Function]

> Returns the minimum of X_1, X_2, . . . , X_N. (MIN) returns the value of MAX.INTEGER (see the Integer Arithmetic section below).

(**MAX** X_1 X_2 . . . X_N) [NoSpread Function]

> Returns the maximum of X_1, X_2, ..., X_N. (MAX) returns the value of MIN.INTEGER (see the Integer Arithmetic section below).

(**ABS** *X*) [Function]

> *X* if *X* > 0, otherwise –*X*. ABS uses GREATERP and MINUS (not IGREATERP and IMINUS).

Integer Arithmetic

The input syntax for an integer is an optional sign (+ or -) followed by a sequence of decimal digits, and terminated by a delimiting character. Integers entered with this syntax are interpreted as decimal integers. Integers in other radices can be entered as follows:

> 123Q
> #o123 If an integer is followed by the letter Q, or preceded by a pound sign and the letter "o", the digits are interpreted as an octal (base 8) integer.

`#b10101`   If an integer is preceded by a pound sign and the letter "b", the digits are interpreted as a binary (base 2) integer.

`#x1A90`   If an integer is preceded by a pound sign and the letter "x", the digits are interpreted as a hexadecimal (base 16) integer.

`#5r1243`   If an integer is preceded by a pound sign, a positive decimal integer BASE, and the letter "r", the digits are interpreted as an integer in the base BASE. For example, `#8r123 = 123Q`, and `#16r12A3 = #x12A3`. When typing a number in a radix above ten, the uppercase letters A through Z can be used as the digits after 9 (but there is no digit above Z, so it is not possible to type all base-99 digits).

Medley keeps no record of how you typed a number, so `77Q` and `63` both correspond to the same integer, and are indistinguishable internally. The function `RADIX` (see Chapter 25), sets the radix used to print integers.

`PACK` and `MKATOM` create numbers when given a sequence of characters observing the above syntax, e.g. `(PACK '(1 2 Q)) => 10`. Integers are also created as a result of arithmetic operations.

The range of integers of various types is implementation-dependent. This information is accessible to you through the following variables:

MIN.SMALLP   [Variable]
MAX.SMALLP   [Variable]

   The smallest/largest possible small integer.

MIN.FIXP   [Variable]
MAX.FIXP   [Variable]

   The smallest/largest possible large integer.

MIN.INTEGER   [Variable]
MAX.INTEGER   [Variable]

   The value of `MAX.INTEGER` and `MIN.INTEGER` are two special system datatypes. For some algorithms, it is useful to have an integer that is larger than any other integer. Therefore, the values of `MAX.INTEGER` and `MIN.INTEGER` are two special data types; the value of `MAX.INTEGER` is `GREATERP` than any other integer, and the value of `MIN.INTEGER` is `LESSP` than any other integer. Trying to do arithmetic using these special bignums, other than comparison, will cause an error.

All of the functions described below work on integers. Unless specified otherwise, if given a floating-point number, they first convert the number to an integer by truncating the fractional bits, e.g., `(IPLUS 2.3 3.8) = 5`; if given a non-numeric argument, they generate an error, `Non-numeric arg`.

(**IPLUS** $X_1 X_2$ ... X_N)   [NoSpread Function]

   Returns the sum $X_1 + X_2 + ... + X_N$. `(IPLUS) = 0`.

(**IMINUS** X)   [Function]

   $-X$

(**IDIFFERENCE** X Y) [Function]

    X – Y

(**ADD1** X) [Function]

    X + 1

(**SUB1** X) [Function]

    X – 1

(**ITIMES** X_1 X_2 ... X_N) [NoSpread Function]

    Returns the product X_1 * X_2 * ... * X_N. (ITIMES) = 1.

(**IQUOTIENT** X Y) [Function]

    X / Y truncated. Examples:

```
(IQUOTIENT 3 2) => 1
(IQUOTIENT -3 2) => -1
```

    If Y is zero, the result is determined by the function OVERFLOW .

(**IREMAINDER** X Y) [Function]

    Returns the remainder when X is divided by Y. Example:

```
(IREMAINDER 5 2) => 1
```

(**IMOD** X N) [Function]

    Computes the integer modulus of X mod N; this differs from IREMAINDER in that the result is always a non-negative integer in the range [0,N).

(**IGREATERP** X Y) [Function]

    T, if $X > Y$; NIL otherwise.

(**ILESSP** X Y) [Function]

    T, if $X < Y$; NIL otherwise.

(**IGEQ** X Y) [Function]

    T, if $X >= Y$; NIL otherwise.

(**ILEQ** X Y) [Function]

    T, if $X <= Y$; NIL otherwise.

(**IMIN** X_1 X_2 ... X_N) [NoSpread Function]

    Returns the minimum of X_1, X_2, ..., X_N. (IMIN) returns the largest possible large integer, the value of MAX.INTEGER.

(**IMAX** X_1 X_2 ... X_N) [NoSpread Function]

 Returns the maximum of X_1, X_2, ..., X_N. (IMAX) returns the smallest possible large integer, the value of MIN.INTEGER.

(**IEQP** X Y) [Function]

 Returns T if X and Y are equal integers; NIL otherwise. Note that EQ may be used if X and Y are known to be small integers. IEQP converts X and Y to integers, e.g., (IEQP 2000 2000.3) => T.

(**FIX** N) [Function]

 If N is an integer, returns N. Otherwise, converts N to an integer by truncating fractional bits For example, (FIX 2.3) => 2, (FIX –1.7) => –1.

 Since FIX is also a programmer's assistant command (see Chapter 13), typing FIX directly to a Medley executive will not cause the function FIX to be called.

(**FIXR** N) [Function]

 If N is an integer, returns N. Otherwise, converts N to an integer by rounding. FIXR will round towards the even number if N is exactly half way between two integers. For example, (FIXR 2.3) => 2, (FIXR –1.7) => –2, (FIXR 3.5) => 4).

(**GCD** N_1 N_2) [Function]

 Returns the greatest common divisor of N_1 and N_2, (GCD 72 64)=8.

Logical Arithmetic Functions

(**LOGAND** X_1 X_2 ... X_N) [NoSpread Function]

 Returns the logical AND of all its arguments, as an integer. Example:

        (LOGAND 7 5 6) => 4

(**LOGOR** X_1 X_2 ... X_N) [NoSpread Function]

 Returns the logical OR of all its arguments, as an integer. Example:

        (LOGOR 1 3 9) => 11

(**LOGXOR** X_1 X_2 ... X_N) [NoSpread Function]

 Returns the logical exclusive OR of its arguments, as an integer. Example:

        (LOGXOR 11 5) => 14
        (LOGXOR 11 5 9) = (LOGXOR 14 9) => 7

(**LSH** X N) [Function]

 (Arithmetic) "Left Shift." Returns X shifted left N places, with the sign bit unaffected. X can be positive or negative. If N is negative, X is shifted right $–N$ places.

(**RSH** *X N*) [Function]

(Arithmetic) "Right Shift." Returns *X* shifted right *N* places, with the sign bit unaffected, and copies of the sign bit shifted into the leftmost bit. *X* can be positive or negative. If *N* is negative, *X* is shifted left *-N* places.

Warning: Be careful if using RSH to simulate division; RSHing a negative number isn't the same as dividing by a power of two.

(**LLSH** *X N*) [Function]
(**LRSH** *X N*) [Function]

"Logical Left Shift" and "Logical Right Shift". The difference between a logical and arithmetic right shift lies in the treatment of the sign bit. Logical shifting treats it just like any other bit; arithmetic shifting will not change it, and will "propagate" rightward when actually shifting rightwards. Note that shifting (arithmetic) a negative number "all the way" to the right yields –1, not 0.

Note: LLSH and LRSH always operate mod-2^{32} arithmetic. Passing a bignum to either of these will cause an error. LRSH of negative numbers will shift 0s into the high bits.

(**INTEGERLENGTH** *X*) [Function]

Returns the number of bits needed to represent *X*. This is equivalent to: 1+floor[log2[abs[*X*]]]. (INTEGERLENGTH 0) = 0.

(**POWEROFTWOP** *X*) [Function]

Returns non-NIL if *X* (coerced to an integer) is a power of two.

(**EVENP** *X Y*) [NoSpread Function]

If *Y* is not given, equivalent to (ZEROP (IMOD *X* 2)); otherwise equivalent to (ZEROP (IMOD X Y)).

(**ODDP** *N MODULUS*) [NoSpread Function]

Equivalent to (NOT (EVENP N MODULUS)). *MODULUS* defaults to 2.

(**LOGNOT** *N*) [Macro]

Logical negation of the bits in *N*. Equivalent to (LOGXOR *N* –1).

(**BITTEST** *N MASK*) [Macro]

Returns T if any of the bits in *MASK* are on in the number *N*. Equivalent to (NOT (ZEROP (LOGAND *N MASK*))).

(**BITCLEAR** *N MASK*) [Macro]

Turns off bits from *MASK* in *N*. Equivalent to (LOGAND *N* (LOGNOT *MASK*)).

(**BITSET** *N MASK*) [Macro]

Turns on the bits from *MASK* in *N*. Equivalent to (LOGOR *N MASK*).

(**MASK.1'S** *POSITION SIZE*) [Macro]

> Returns a bit-mask with *SIZE* one-bits starting with the bit at *POSITION*. Equivalent to (LLSH (SUB1 (EXPT 2 *SIZE*)) *POSITION*).

(**MASK.0'S** *POSITION SIZE*) [Macro]

> Returns a bit-mask with all one bits, except for *SIZE* bits starting at *POSITION*. Equivalent to (LOGNOT (MASK.1'S *POSITION SIZE*)).

(**LOADBYTE** *N POS SIZE*) [Function]

> Extracts *SIZE* bits from *N*, starting at position *POS*. Equivalent to (LOGAND (RSH *N POS*) (MASK.1'S 0 *SIZE*)).

(**DEPOSITBYTE** *N POS SIZE VAL*) [Function]

> Insert *SIZE* bits of *VAL* at position *POS* into *N*, returning the result. Equivalent to

        (LOGOR (BITCLEAR N (MASK.1'S POS SIZE))
          (LSH (LOGAND VAL (MASK.1'S 0 SIZE))
            POS))

(**ROT** *X N FIELDSIZE*) [Function]

> "Rotate bits in field". It performs a bitwise left-rotation of the integer *X*, by *N* places, within a field of *FIELDSIZE* bits wide. Bits being shifted out of the position selected by (EXPT 2 (SUB1 *FIELDSIZE*)) will flow into the "units" position.

The notions of position and size can be combined to make up a "byte specifier", which is constructed by the macro BYTE [note reversal of arguments as compared with the above functions]:

(**BYTE** *SIZE POSITION*) [Macro]

> Constructs and returns a "byte specifier" containing *SIZE* and *POSITION*.

(**BYTESIZE** *BYTESPEC*) [Macro]

> Returns the SIZE componant of the "byte specifier" *BYTESPEC*.

(**BYTEPOSITION** *BYTESPEC*) [Macro]

> Returns the POSITION componant of the "byte specifier" *BYTESPEC*.

(**LDB** *BYTESPEC VAL*) [Macro]

> Equivalent to

> (LOADBYTE *VAL* (BYTEPOSITION *BYTESPEC*)(BYTESIZE *BYTESPEC*))

(**DPB** *N BYTESPEC VAL*) [Macro]

> Equivalent to

> (DEPOSITBYTE *VAL* (BYTEPOSITION *BYTESPEC*)(BYTESIZE *BYTESPEC*) *N*)

Floating-Point Arithmetic

A floating-point number is input as a signed integer, followed by a decimal point, and another sequence of digits called the fraction, followed by an exponent (represented by E followed by a signed integer) and terminated by a delimiter.

Both signs are optional, and either the fraction following the decimal point, or the integer preceding the decimal point may be omitted. One or the other of the decimal point or exponent may also be omitted, but at least one of them must be present to distinguish a floating-point number from an integer. For example, the following will be recognized as floating-point numbers:

```
5.       5.00    5.01    .3
5E2      5.1E2   5E-3    -5.2E+6
```

Floating-point numbers are printed using the format control specified by the function FLTFMT (see Chapter 25). FLTFMT is initialized to T, or free format. For example, the above floating-point numbers would be printed free format as:

```
5.0      5.0     5.01    .3
500.0    510.0   .005    -5.2E6
```

Floating-point numbers are created by the reader when a "." or an E appears in a number, e.g., 1000 is an integer, 1000. a floating-point number, as are 1E3 and 1.E3. Note that 1000D, 1000F, and 1E3D are perfectly legal literal atoms. Floating-point numbers are also created by PACK and MKATOM, and as a result of arithmetic operations.

PRINTNUM (see Chapter 25) permits greater control over the printed appearance of floating-point numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

The floating-point number range is stored in the following variables:

MIN.FLOAT   [Variable]

    The smallest possible floating-point number.

MAX.FLOAT   [Variable]

    The largest possible floating-point number.

All of the functions described below work on floating-point numbers. Unless specified otherwise, if given an integer, they first convert the number to a floating-point number, e.g., (FPLUS 1 2.3) <=> (FPLUS 1.0 2.3) => 3.3; if given a non-numeric argument, they generate an error, Non-numeric arg.

(**FPLUS** $X_1 X_2 \ldots X_N$)   [NoSpread Function]

    $X_1 + X_2 + \ldots + X_N$

(**FMINUS** X)   [Function]

    $-X$

(**FDIFFERENCE** $X\,Y$)   [Function]

    $X - Y$

(**FTIMES** $X_1 X_2 \ldots X_N$)   [NoSpread Function]

    $X_1 * X_2 * \ldots * X_N$

(**FQUOTIENT** $X Y$)   [Function]

    X / Y.

The results of division by zero and floating-point overflow is determined by the function OVERFLOW.

(**FREMAINDER** $X Y$)   [Function]

Returns the remainder when X is divided by Y. Equivalent to:

    (FDIFFERENCE X (FTIMES Y (FIX (FQUOTIENT $X Y$))))

Example:

    (FREMAINDER 7.5 2.3) => 0.6

(**FGREATERP** $X Y$)   [Function]

T, if $X > Y$, NIL otherwise.

(**FLESSP** $X Y$)   [Function]

T, if $X < Y$, NIL otherwise.

(**FEQP** $X Y$)   [Function]

Returns T if X and Y are equal floating-point numbers; NIL otherwise. FEQP converts X and Y to floating-point numbers.

(**FMIN** $X_1 X_2 \ldots X_N$)   [NoSpread Function]

Returns the minimum of X_1, X_2, $\ldots$, X_N. (FMIN) returns the largest possible floating-point number, the value of MAX.FLOAT.

(**FMAX** $X_1 X_2 \ldots X_N$)   [NoSpread Function]

Returns the maximum of X_1, X_2, $\ldots$, X_N. (FMAX) returns the smallest possible floating-point number, the value of MIN.FLOAT.

(**FLOAT** X)   [Function]

Converts X to a floating-point number. Example:

    (FLOAT 0) => 0.0

Transcendental Arithmetic Functions

(**EXPT** $A N$)   [Function]

Returns A^N. If A is an integer and N is a positive integer, returns an integer, e.g, (EXPT 3 4) => 81, otherwise returns a floating-point number. If A is negative and N fractional,

generates the error, `Illegal exponentiation`. If *N* is floating and either too large or too small, generates the error, `Value out of range expt`.

(**SQRT** *N*)   [Function]

Returns the square root of *N* as a floating-point number. *N* may be fixed or floating-point. Generates an error if *N* is negative.

(**LOG** *X*)   [Function]

Returns the natural logarithm of *X* as a floating-point number. *X* can be integer or floating-point.

(**ANTILOG** *X*)   [Function]

Returns the floating-point number whose logarithm is *X*. *X* can be integer or floating-point. Example:

        (ANTILOG 1) = e => 2.71828...

(**SIN** *X RADIANSFLG*)   [Function]

Returns the sine of *X* as a floating-point number. *X* is in degrees unless *RADIANSFLG* = T.

(**COS** *X RADIANSFLG*)   [Function]

Similar to `SIN`.

(**TAN** *X RADIANSFLG*)   [Function]

Similar to `SIN`.

(**ARCSIN** *X RADIANSFLG*)   [Function]

The value of `ARCSIN` is a floating-point number, and is in degrees unless *RADIANSFLG* = T. In other words, if (ARCSIN *X RADIANSFLG*) = Z then (SIN Z *RADIANSFLG*) = *X*. The range of the value of `ARCSIN` is -90 to +90 for degrees, $-\pi/2$ to $\pi/2$ for radians. *X* must be a number between -1 and 1.

(**ARCCOS** *X RADIANSFLG*)   [Function]

Similar to `ARCSIN`. Range is 0 to 180, 0 to π.

(**ARCTAN** *X RADIANSFLG*)   [Function]

Similar to `ARCSIN`. Range is 0 to 180, 0 to π.

(**ARCTAN2** *Y X RADIANSFLG*)   [Function]

Computes (ARCTAN (FQUOTIENT *Y X*) *RADIANSFLG*), and returns a corresponding value in the range −180 to 180 (or $-\pi$ to π), i.e. the result is in the proper quadrant as determined by the signs of *X* and *Y*.

Generating Random Numbers

(**RAND** *LOWER UPPER*)   [Function]

> Returns a pseudo-random number between *LOWER* and *UPPER* inclusive, i.e., RAND can be used to generate a sequence of random numbers. If both limits are integers, the value of RAND is an integer, otherwise it is a floating-point number. The algorithm is completely deterministic, i.e., given the same initial state, RAND produces the same sequence of values. The internal state of RAND is initialized using the function RANDSET.

(**RANDSET** *X*)   [Function]

> Returns the internal state of RAND. If *X* = NIL, just returns the current state. If *X* = T, RAND is initialized using the clocks, and RANDSET returns the new state. Otherwise, *X* is interpreted as a previous internal state, i.e., a value of RANDSET, and is used to reset RAND. For example,

```
←(SETQ OLDSTATE (RANDSET))
 ...
←(for X from 1 to 10 do (PRIN1 (RAND 1 10)))
 2847592748NIL
←(RANDSET OLDSTATE)
 ...
←(for X from 1 to 10 do (PRIN1 (RAND 1 10)))
 2847592748NIL
```

INTERLISP-D REFERENCE MANUAL

[This page intentionally left blank]

Hiding the details of your code makes it more readable, and lets you program more efficiently. Data structures are a good example: You're better off if you can say "Fetch me the SPEED field from this AIRPLANE" rather than having to say (CAR (CDDDR (CADR AIRPLANE))). You can declare data structures used by your programs, then work with field names rather than access details. Using the declarations, Medley performs the access/storage operations you request. If you change a data structure's declaration, your programs automatically adjust.

You describe the format of a data structure (record) by making a "record declaration" (see the Record Declarations section below). The record declaration is a description of the record, associating names with its various parts, or "fields". For example, the record declaration

```
(RECORD MSG (FROM TO TEXT))
```

describes a data structure called MSG, that has three fields: FROM, TO, and TEXT. You can refer to these fields by name, to get their values or to store new values into them, by using FETCH and REPLACE:

```
(fetch (MSG FROM)of MYMSG)
(replace (MSG TO) of MYMSG with "John Doe")
```

You create new MSGs with CREATE:

```
(SETQ MYMSG (create MSG))
```

and TYPE? tells you whether some object is a MSG:

```
(IF (TYPE? MSG THIS-THING) then (SEND-MSG THIS-THING))
```

So far we've said nothing about *how* your MSG is represented—when you're writing FETCHes and REPLACEs, it doesn't matter. But you *can* control the representation: The symbol RECORD in the declaration above causes each MSG to be represented as a list. There are a number of options, up to creating a completely new Lisp data type; each has its own specifier symbol, and they're described in detail below.

The record package is implemented using DWIM and CLISP, so it will do spelling correction on field names, record types, etc. Record operations are translated using all CLISP declarations in effect (standard/fast/undoable).

The file manager's RECORDS command lets you give record declarations (see Chapter 17), and FILES? and CLEANUP will tell you about record declarations that need to be dumped.

FETCH and REPLACE

The fields of a record are accessed and changed with FETCH and REPLACE. If x is a MSG data structure, (fetch FROM of X) will return the value of the FROM field of x, and (replace FROM of X with Y) will replace this field with the value of Y. In general, the value of a REPLACE operation is the same as the value stored into the field.

Note that (fetch FROM of X) assumes that x is an instance of the record MSG—the interpretation of (fetch FROM of X) never depends on the *value* of x. If x is not a MSG, this may produce incorrect results.

If there is another record declaration, (RECORD REPLY (TEXT RESPONSE)), then (fetch TEXT of X) is ambiguous, because x could be either a MSG or a REPLY record. In this case, an error will occur, Ambiguous record field. To clarify this, give FETCH and REPLACE a list for their "field" argument: (fetch (MSG TEXT) of X) will fetch the TEXT field of a MSG record. If a field has an *identical* interpretation in two declarations, e.g., if the field

TEXT occurred in the same location within the declarations of MSG and REPLY, then (fetch TEXT of X) would *not* be ambiguous.

If there's a conflict, "user" record declarations take precedence over "system" record declarations. System records are declared by including (SYSTEM) in the declaration (see the Record Declarations section below). All of the records defined in the standard Medley system are system records.

Another complication can occur if the fields of a record are themselves records. The fields of a record can be further broken down into sub-fields by a "subdeclaration" within the record declaration. For example,

> (RECORD NODE (POSITION . LABEL) (RECORD POSITION (XLOC . YLOC)))

lets you access the POSITION field with (fetch POSITION of X), or its subfield XLOC with (fetch XLOC of X).

You may also declare that field name in a *separate* record declaration. For instance, the TEXT field in the MSG and REPLY records above may be subdivided with the seperate record declaration (RECORD TEXT (HEADER TXT)). You get to fields of subfields (to any level of nesting) by specifying the "data path" as a list of record/field names, where there is some path from each record to the next in the list. For instance,

> (fetch (MSG TEXT HEADER) of X)

treats X as a MSG record, fetches its TEXT field, and fetches *its* HEADER field. You only need to give enough of the data path to disambiguate it. In this case, (fetch (MSG HEADER) of X) is sufficient: Medley searches among all current record declarations for a path from each name to the next, considering first local declarations (see Chapter 21) and then global ones. Of course, if you had two records with HEADER fields, you get an Ambiguous data path error.

FETCH and REPLACE are translated using the CLISP declarations in effect (see Chapter 21). FFETCH and FREPLACE are fast versions that don't do any type checking. /REPLACE insures undoable declarations.

Record Declarations

You define records by evaluating declarations of the form:

> (*RECORD-TYPE RECORD-NAME RECORD-FIELDS . RECORD-TAIL*)

RECORD-TYPE specifies the "type" of data you're declaring, and controls how instances will be stored internally. The different record types are described below.

RECORD-NAME is a symbol used to identify the record declaration for CREATE, TYPE?, FETCH and REPLACE, and dumping to files (see Chapter 17). DATATYPE and TYPERECORD declarations also use *RECORD-NAME* to identify the data structure (as described below).

RECORD-FIELDS describes the structure of the record. Its exact interpretation varies with RECORD-TYPE. Generally, it names the fields within the record that can be accessed with FETCH and REPLACE.

RECORD-TAIL is an optional list where you can specify default values for record fields, special CREATE and TYPE? forms, and subdeclarations (described below).

Record declarations are Lisp programs, and could be included in functions, changing a record declaration at run-time. *Don't do it.* You risk creating a structure with one declaration, and trying to fetch from it with another—complete chaos results. If you need to change record declarations dynamically, consider using association lists or property lists.

Record Types

The `RECORD-TYPE` field of the record declaration specifies how the data object is created, and how the various record fields are accessed. Depending on the record type, the record fields may be stored in a list, or in an array, or on a symbol's property list. The following record types are defined:

RECORD [Record Type]

>The fields of a RECORD are kept in a list. `RECORD-FIELDS` is a list; each non-NIL symbol is a field-name to be associated with the corresponding element or tail of a list structure. For example, with the declaration (RECORD MSG (FROM TO . TEXT)), (fetch FROM of X) translates as (CAR X).

>NIL can be used as a place marker for an unnamed field, e.g., (A NIL B) describes a three element list, with B corresponding to the third element. A number may be used to indicate a sequence of NILs, e.g. (A 4 B) is interpreted as (A NIL NIL NIL NIL B).

DATATYPE [Record Type]

>Defines a new user data type with type name `RECORD-NAME`. Unlike other record types, the instances of a DATATYPE are represented with a completely new Lisp type, and not in terms of other existing types.

>`RECORD-FIELDS` is a list of field specifications, where each specification is either a list (`FIELDNAME  FIELDTYPE`), or an symbol `FIELDNAME`. If `FIELDTYPE` is omitted, it defaults to POINTER. Possible values for `FIELDTYPE` are:

>>POINTER Field contains a pointer to any arbitrary Interlisp object.

>>INTEGER
>>FIXP Field contains a signed integer. Caution: An INTEGER field is not capable of holding everything that satisfies FIXP, such as bignums.

>>FLOATING
>>FLOATP Field contains a floating point number.

>>SIGNEDWORD Field contains a 16-bit signed integer.

>>FLAG Field is a one bit field that "contains" T or NIL.

>>BITS *N* Field contains an *N*-bit unsigned integer.

>>BYTE Equivalent to BITS 8.

>>WORD Equivalent to BITS 16.

>>XPOINTER Field contains a pointer like POINTER, but the field is *not* reference counted by the garbage collector. XPOINTER fields are useful for implementing back-pointers in structures that would be circular and not otherwise collected by the reference-counting garbage collector.

>>**Warning**: Use XPOINTER fields with great care. You can damage the integrity of the storage allocation system by using pointers to objects that have been garbage collected. Code that uses XPOINTER fields should be sure that the objects pointed to have not been garbage collected. This can be done in two ways: The first is to maintain the object in a global structure,

so that it is never garbage collected until explicitly deleted from the structure, at which point the program must invalidate all the ₓₚₒₗₙₜₑᵣ fields of other objects pointing at it. The second is to declare the object as a ᴅᴀᴛᴀᴛʏᴘᴇ beginning with a ᴘₒₗₙₜₑᵣ field that the program maintains as a pointer to an object of another type (e.g., the object containing the ₓₚₒₗₙₜₑᵣ pointing back at it), and test that field for reasonableness whenever using the contents of the ₓₚₒₗₙₜₑᵣ field.

For example, the declaration

```
(DATATYPE FOO
    ((FLG BITS 12) TEXT HEAD (DATE BITS 18)
    (PRIO FLOATP) (READ? FLAG)))
```

would define a data type ꜰₒₒ with two pointer fields, a floating point number, and fields for a ₁₂ and ₁₈ bit unsigned integers, and a flag (one bit). Fields are allocated in such a way as to optimize the storage used and not necessarily in the order specified. Generally, a ᴅᴀᴛᴀᴛʏᴘᴇ record is much more storage compact than the corresponding ᴿᴱᴄᴼᴿᴰ structure would be; in addition, access is faster.

Since the user data type must be set up at *run*-time, the ᴿᴱᴄᴼᴿᴰˢ file package command will dump a ᴅᴇᴄʟᴀʀᴇᴅᴀᴛᴀᴛʏᴘᴇ expression as well as the ᴅᴀᴛᴀᴛʏᴘᴇ declaration itself. If the record declaration is otherwise not needed at runtime, it can be kept out of the compiled file by using a (DECLARE: DONTCOPY --) expression (see Chapter 17), but it is still necessary to ensure that the datatype is properly initialized. For this, one can use the ₗₙₗₜᴿᴱᴄᴼᴿᴰˢ file package command (see Chapter 17), which will dump only the ᴅᴇᴄʟᴀʀᴇᴅᴀᴛᴀᴛʏᴘᴇ expression.

Note: When defining a new data type, it is sometimes useful to call the function ᴅᴇꜰᴘʀₗₙₜ (see Chapter 25) to specify how instances of the new data type should be printed. This can be specified in the record declaration by including an ₗₙₗₜ record specification (see the Optional Record Specifications section below), e.g. (DATATYPE QV.TYPE ... (INIT (DEFPRINT 'QV.TYPE (FUNCTION PRINT.QV.TYPE)))).

ᴅᴀᴛᴀᴛʏᴘᴇ declarations cannot be used within local record declarations (see Chapter 21).

TYPERECORD   [Record Type]

Similar to ᴿᴱᴄᴼᴿᴰ, but the record name is added to the front of the list structure to signify what "type" of record it is. This type field is used in the translation of ᴛʏᴘᴇ? expressions. ᴄᴿᴇᴀᴛᴇ will insert an extra field containing *RECORD-NAME* at the beginning of the structure, and the translation of the access and storage functions will take this extra field into account. For example, for (TYPERECORD MSG (FROM TO . TEXT)), (fetch FROM of X) translates as (CADR X), not (CAR X).

ASSOCRECORD   [Record Type]

Describes lists where the fields are stored in association list format:

$$((FIELDNAME_1 . VALUE_1) (FIELDNAME_2 . VALUE_2) ...)$$

RECORD-FIELDS is a list of symbols, the permissable field names in the association list. Access is done with ᴀˢˢₒᴄ (or ꜰᴀˢˢₒᴄ, if the current CLISP declarations are ꜰᴀˢᴛ, see Chapter 21), storing with ᴘᴜᴛᴀˢˢₒᴄ.

PROPRECORD [Record Type]

Describes lists where the fields are stored in property list format:

(*FIELDNAME$_1$ VALUE$_1$ FIELDNAME$_2$ VALUE$_2$* ...)

RECORD-FIELDS is a list of symbols, the permissable field names in the property list. Access is done with LISTGET, storing with LISTPUT.

Both ASSOCRECORD and PROPRECORD are useful for defining data structures where many of the fields are NIL. CREATEing one these record types only stores those fields that are non-NIL. Note, however, that with the record declaration (PROPRECORD FIE (H I J)) the expression (create FIE) would still construct (H NIL), since a later operation of (replace J of X with Y) could not possibly change the instance of the record if it were NIL.

ARRAYRECORD [Record Type]

ARRAYRECORDs are stored as arrays. *RECORD-FIELDS* is a list of field names that are associated with the corresponding elements of an array. NIL can be used as a place marker for an unnamed field (element). Positive integers can be used as abbreviation for the corresponding number of NILs. For example, (ARRAYRECORD (ORG DEST NIL ID 3 TEXT)) describes an eight-element array, with ORG corresponding to the first element, ID to the fourth, and TEXT to the eighth.

ARRAYRECORD only creates arrays of pointers. Other kinds of arrays must be implemented with ACCESSFNS (see below).

HASHLINK [Record Type]

The HASHLINK record type can be used with any type of data object: it specifies that the value of a single field can be accessed by hashing the data object in a given hash array. Since the HASHLINK record type describes an access method, rather than a data structure, CREATE is meaningless for HASHLINK records.

RECORD-FIELDS is either a symbol *FIELD-NAME*, or a list (*FIELD-NAME HARRAYNAME HARRAYSIZE*). *HARRAYNAME* is a variable whose value is the hash array to be used; if not given, SYSHASHARRAY is used. If the value of the variable *HARRAYNAME* is not a hash array (at the time of the record declaration), it will be set to a new hash array with a size of *HARRAYSIZE*. *HARRAYSIZE* defaults to 100.

The HASHLINK record type is useful as a subdeclaration to other records to add additional fields to already existing data structures (see the Optional Record Specifications section below). For example, suppose that FOO is a record declared with (RECORD FOO (A B C)). To add a new field BAR, without modifying the existing data strutures, redeclare FOO with:

(RECORD FOO (A B C) (HASHLINK FOO (BAR BARHARRAY)))

Now, (fetch BAR of X) will translate into (GETHASH X BARHARRAY), hashing off the existing list X.

ATOMRECORD [Record Type]

ATOMRECORDs are stored on the property lists of symbols. *RECORD-FIELDS* is a list of property names. Accessing is performed with GETPROP, storing with PUTPROP. The CREATE expression is not initially defined for ATOMRECORD records.

BLOCKRECORD   [Record Type]

BLOCKRECORD is used in low-level system programming to "overlay" an organized structure over an arbitrary piece of raw storage. *RECORD–FIELDS* is interpreted exactly as with a DATATYPE declaration, except that fields are *not* automatically rearranged to maximize storage efficiency. Like an ACCESSFNS record, a BLOCKRECORD does not have concrete instances; it merely provides a way of interpreting some existing block of storage. So you can't create an instance of a BLOCKRECORD (unless the declaration includes an explicit CREATE expression), nor is there a default type? expression for a BLOCKRECORD.

Warning: Exercise caution in using BLOCKRECORD declarations, as they let you fetch and store arbitrary data in arbitrary locations, thereby evading Medley's normal type system. Except in very specialized situations, a BLOCKRECORD should never contain POINTER or XPOINTER fields, nor be used to overlay an area of storage that contains pointers. Such use could compromise the garbage collector and storage allocation system. You are responsible for ensuring that all FETCH and REPLACE expressions are performed only on suitable objects, as no type testing is performed.

A typical use for a BLOCKRECORD in user code is to overlay a non-pointer portion of an existing DATATYPE. For this use, the LOCF macro is useful. (LOCF (fetch *FIELD* of *DATUM*)) can be used to refer to the storage that begins at the first word that contains *FIELD* of *DATUM*. For example, to define a new kind of Ethernet packet, you could overlay the "body" portion of the ETHERPACKET datatype declaration as follows:

```
(ACCESSFNS MYPACKET
((MYBASE (LOCF (fetch (ETHERPACKET EPBODY) of DATUM))))
 (BLOCKRECORD MYBASE
 ((MYTYPE WORD) (MYLENGTH WORD) (MYSTATUS BYTE)
 (MYERRORCODE BYTE) (MYDATA INTEGER)))
 (TYPE? (type? ETHERPACKET DATUM)))
```

With this declaration in effect, the expression (fetch MYLENGTH of PACKET) would retrieve the second 16-bit field beyond the place inside PACKET where the EPBODY field starts.

ACCESSFNS   [Record Type]

ACCESSFNS lets you specify arbitrary functions to fetch and store data. For each field name, you specify how it is to be accessed and set. This lets you use arbitrary data structures, with complex access methods. Most often, ACCESSFNS are useful when you can compute one field's value from other fields. If you're representing a time period by its start and duration, you could add an ACCESSFNS definition for the ending time that did the obvious addition.

RECORD–FIELDS is a list of elements of the form (*FIELD–NAME ACCESSDEF SETDEF*). *ACCESSDEF* should be a function of one argument, the datum, and will be used for accessing the value of the field. *SETDEF* should be a function of two arguments, the datum and the new value, and will be used for storing a new value in a field. *SETDEF* may be omitted, in which case, no storing operations are allowed.

ACCESSDEF and/or *SETDEF* may also be a form written in terms of variables DATUM and (in *SETDEF*) NEWVALUE. For example, given the declaration

```
[ACCESSFNS FOO
((FIRSTCHAR (NTHCHAR DATUM 1) (RPLSTRING DATUM 1 NEWVALUE)) (RESTCHARS (SUBSTRING DATUM 2]
```

(replace (FOO FIRSTCHAR) of X with Y) would translate to (RPLSTRING X 1 Y). Since no *SETDEF* is given for the RESTCHARS field, attempting to perform (replace (FOO RESTCHARS) of X with Y) would generate an error, Replace undefined for field. Note that ACCESSFNS do not have a CREATE definition. However, you may supply one in the defaults or subdeclarations of the declaration, as described below. Attempting to CREATE an ACCESSFNS record without specifying a create definition will cause an error Create not defined for this record.

ACCESSDEF and *SETDEF* can also be a property list which specify FAST, STANDARD and UNDOABLE versions of the ACCESSFNS forms, e.g.

```
[ACCESSFNS LITATOM
        ((DEF (STANDARD GETD FAST FGETD)
              (STANDARD PUTD UNDOABLE /PUTD]
```

means if FAST declaration is in effect, use FGETD for fetching, if UNDOABLE, use /PUTD for saving (see CLISP declarations, see Chapter 21).

SETDEF forms should be written so that they return the new value, to be consistant with REPLACE operations for other record types. The REPLACE does not enforce this, though.

ACCESSFNS let you use data structures not specified by one of the built-in record types. For example, one possible representation of a data structure is to store the fields in *parallel* arrays, especially if the number of instances required is known, and they needn't be garbage collected. To implement LINK with two fields FROM and TO, you'd have two arrays FROMARRAY and TOARRAY. The representation of an "instance" of LINK would be an integer, used to index into the arrays. This can be accomplished with the declaration:

```
[ACCESSFNS LINK
    ((FROM (ELT FROMARRAY DATUM)
           (SETA FROMARRAY DATUM NEWVALUE))
     (TO (ELT TOARRAY DATUM)
         (SETA TOARRAY DATUM NEWVALUE)))
    (CREATE (PROG1 (SETQ LINKCNT (ADD1 LINKCNT))
                   (SETA FROMARRAY LINKCNT FROM)
                   (SETA TOARRAY LINKCNT TO)))
    (INIT (PROGN
              (SETQ FROMARRAY (ARRAY 100))
              (SETQ TOARRAY (ARRAY 100))
              (SETQ LINKCNT 0)]
```

To create a new LINK, a counter is incremented and the new elements stored. (Note: The CREATE form given the declaration probably should include a test for overflow.)

Optional Record Specifications

After the *RECORD-FIELDS* item in a record declaration expression there can be an arbitrary number of additional expressions in *RECORD-TAIL*. These expressions can be used to specify default values for record fields, special CREATE and TYPE? forms, and subdeclarations. The following expressions are permitted:

FIELD-NAME ← *FORM*   Allows you to specify within the record declaration the default value to be stored in *FIELD-NAME* by a CREATE (if no value is given within the CREATE expression itself). Note that *FORM* is evaluated at CREATE time, not when the declaration is made.

(CREATE *FORM*)   Defines the manner in which CREATE of this record should be performed. This provides a way of specifying how ACCESSFNS should be created or overriding the usual definition of CREATE. If *FORM* contains the field-names of the declaration as variables, the forms given in the

CREATE operation will be substituted in. If the word DATUM appears in the create form, the *original* CREATE definition is inserted. This effectively allows you to "advise" the create.

(INIT *FORM*)   Specifies that *FORM* should be evaluated when the record is declared. *FORM* will also be dumped by the INITRECORDS file package command (see Chapter 17).

For example, see the example of an ACCESSFNS record declaration above. In this example, FROMARRAY and TOARRAY are initialized with an INIT form.

(TYPE? *FORM*)   Defines the manner in which TYPE? expressions are to be translated. *FORM* may either be an expression in terms of DATUM or a function of one argument.

(SUBRECORD *NAME* .
DEFAULTS)   *NAME* must be a field that appears in the current declaration and the name of another record. This says that, for the purposes of translating CREATE expressions, substitute the top-level declaration of *NAME* for the SUBRECORD form, adding on any defaults specified.

For example: Given (RECORD B (E F G)), (RECORD A (B C D) (SUBRECORD B)) would be treated like (RECORD A (B C D) (RECORD B (E F G))) for the purposes of translating CREATE expressions.

a subdeclaration   If a record declaration expression occurs among the record specifications of another record declaration, it is known as a "subdeclaration." Subdeclarations are used to declare that fields of a record are to be interpreted as another type of record, or that the record data object is to be interpreted in more than one way.

The *RECORD–NAME* of a subdeclaration must be either the *RECORD–NAME* of its immediately superior declaration or one of the superior's field-names. Instead of identifying the declaration as with top level declarations, the record-name of a subdeclaration identifies the parent field or record that is being described by the subdeclaration. Subdeclarations can be nested to an arbitrary depth.

Giving a subdeclaration (RECORD *NAME₁* *NAME₂*) is a simple way of defining a *synonym* for the field *NAME₁*.

It is possible for a given field to have more than one subdeclaration. For example, in

(RECORD FOO (A B) (RECORD A (C D)) (RECORD A (Q R)))

(Q R) and (C D) are "overlayed," i.e. (fetch Q of X) and (fetch C of X) would be equivalent. In such cases, the *first* subdeclaration is the one used by CREATE.

(SYNONYM *FIELD*

(*SYN₁* ... *SYNₙ*)) *FIELD* must be a field that appears in the current declaration. This defines *SYN₁* ... *SYNₙ* all as synonyms of *FIELD*. If there is only one synonym, this can be written as (SYNONYM *FIELD SYN*).

(SYSTEM) If (SYSTEM) is included in a record declaration, this indicates that the record is a "system" record rather than a "user" record. The only distinction between the two types of records is that "user" record declarations take precedence over "system" record declarations, in cases where an unqualified field name would be considered ambiguous. All of the records defined in the standard Medley system are defined as system records.

CREATE

You can create RECORDs by hand if you like, using CONS, LIST, etc. But that defeats the whole point of hiding implementation details. So much easier to use:

    (create *RECORD-NAME . ASSIGNMENTS*)

CREATE translates into an appropriate Interlisp form that uses CONS, LIST, PUTHASH, ARRAY, etc., to create the new datum with the its fields initialized to the values you specify. *ASSIGNMENTS* is optional and may contain expressions of the following form:

FIELD-NAME ← *FORM* Specifies initial value for *FIELD-NAME*.

USING *FORM* *FORM* is an existing instance of *RECORD-NAME*. If you don't specify a value for some field, the value of the corresponding field in *FORM* is to be used.

COPYING *FORM* Like USING, but the corresponding values are copied (with COPYALL).

REUSING *FORM* Like USING, but wherever possible, the corresponding *structure* in *FORM* is used.

SMASHING *FORM* A new instance of the record is not created at all; rather, new field values are smashed into *FORM*, which CREATE then returns.

When it makes a difference, Medley goes to great pains to make its translation do things in the same order as the original CREATE expression. For example, given the declaration (RECORD CONS (CAR . CDR)), the expression (create CONS CDR←X CAR←Y) will translate to (CONS Y X), but (create CONS CDR←(FOO) CAR←(FIE)) will translate to ((LAMBDA ($$1) (CONS (PROGN (SETQ $$1 (FOO)) (FIE)) $$1))) because FOO might set some variables used by FIE.

How are USING and REUSING different? (create *RECORD* reusing *FORM* ...) doesn't do any destructive operations on the value of *FORM*, but *will* incorporate as much as possible of the old data structure into the new one. On the other hand, (create *RECORD* using *FORM* ...) will create a completely new data structure, with only the *contents* of the fields re-used. For example, REUSING a PROPRECORD just CONSes the new property names and values onto the list, while USING copies the top level of the list. Another example of this distinction occurs when a field is elaborated by a subdeclaration: USING will create a new instance of the sub-record, while REUSING will use the old contents of the field (unless some field of the subdeclaration is assigned in the CREATE expression.)

If the value of a field is neither explicitly specified, nor implicitly specified via USING, COPYING or REUSING, the default value in the declaration is used, if any, otherwise NIL. (For BETWEEN fields in DATATYPE records, N_1 is used; for other non-pointer fields zero is used.) For example, following (RECORD A (B C D) D ← 3)

```
(create A B ← T) ==> (LIST T NIL 3)
(create A B ← T using X) ==> (LIST T (CADR X) (CADDR X))
(create A B ← T copying X)) ==> [LIST T (COPYALL (CADR X)) (COPYALL (CADDR X]
(create A B ← T reusing X) ==> (CONS T (CDR X))
```

TYPE?

The record package allows you to test if a given datum "looks like" an instance of a record. This can be done via an expression of the form (type? *RECORD-NAME FORM*).

TYPE? is mainly intended for records with a record type of DATATYPE or TYPERECORD. For DATATYPEs, the TYPE? check is exact; i.e. the TYPE? expression will return non-NIL only if the value of *FORM* is an instance of the record named by *RECORD-NAME*. For TYPERECORDs, the TYPE? expression will check that the value of *FORM* is a list beginning with *RECORD-NAME*. For ARRAYRECORDs, it checks that the value is an array of the correct size. For PROPRECORDs and ASSOCRECORDs, a TYPE? expression will make sure that the value of *FORM* is a property/association list with property names among the field-names of the declaration.

There is no built-in type test for records of type ACCESSFNS, HASHLINK or RECORD. Type tests can be defined for these kinds of records, or redefined for the other kinds, by including an expression of the form (TYPE? *COM*) in the record declaration (see the Record Declarations section below). Attempting to execute a TYPE? expression for a record that has no type test causes an error, Type? not implemented for this record.

WITH

Often one wants to write a complex expression that manipulates several fields of a single record. The WITH construct can make it easier to write such expressions by allowing one to refer to the fields of a record as if they were variables within a lexical scope:

    (with *RECORD-NAME RECORD-INSTANCE FORM$_1$ ... FORM$_N$*)

RECORD-NAME is the name of a record, and *RECORD-INSTANCE* is an expression which evaluates to an instance of that record. The expressions *FORM$_1$ ... FORM$_N$* are evaluated so that references to variables which are field-names of *RECORD-NAME* are implemented via FETCH and SETQs of those variables are implemented via REPLACE.

For example, given

```
(RECORD RECN (FLD1 FLD2))
(SETQ INST (create RECN FLD1 ← 10 FLD2 ← 20))
```

Then the construct

```
(with RECN INST (SETQ FLD2 (PLUS FLD1 FLD2]
```

is equivalent to

```
(replace FLD2 of INST with (PLUS (fetch FLD1 of INST) (fetch FLD2 of INST]
```

Warning: WITH is implemented by doing simple substitutions in the body of the forms, without regard for how the record fields are used. This means, for example, if the record FOO is defined by (RECORD FOO (POINTER1 POINTER2)), then the form

```
(with FOO X (SELECTQ Y (POINTER1 POINTER1) NIL]
```

will be translated as

```
(SELECTQ Y ((CAR X) (CAR X)) NIL]
```

Be careful that record field names are not used except as variables in the WITH forms.

Defining New Record Types

In addition to the built-in record types, you can declare your own record types by performing the following steps:

1. Add the new record-type to the value of CLISPRECORDTYPES.

2. Perform (MOVD 'RECORD *RECORD-TYPE*).

3. Put the name of a function which will return the translation on the property list of *RECORD–TYPE*, as the value of the property USERRECORDTYPE. Whenever a record declaration of type *RECORD–TYPE* is encountered, this function will be passed the record declaration as its argument, and should return a *new* record declaration which the record package will then use in its place.

Manipulating Record Declarations

(**EDITREC** *NAME COM₁ ... COMₙ*)   [NLambda NoSpread Function]

EDITREC calls the editor on a copy of all declarations in which *NAME* is the record name or a field name. On exit, it redeclares those that have changed and undeclares any that have been deleted. If *NAME* is NIL, *all* declarations are edited.

$COM_1 \ldots COM_N$ are (optional) edit commands.

When you redeclare a global record, the translations of all expressions involving that record or any of its fields are automatically deleted from CLISPARRAY, and thus will be recomputed using the new information. If you change a *local* record declaration (see Chapter 21), or change some other CLISP declaration (see Chapter 21), e.g., STANDARD to FAST, and wish the new information to affect record expressions already translated, you must make sure the corresponding translations are removed, usually either by CLISPIFYing or using the DW edit macro.

(**RECLOOK** *RECNAME — — — —*)   [Function]

Returns the entire declaration for the record named *RECNAME*; NIL if there is no record declaration with name *RECNAME*. Note that the record package maintains internal state about current record declarations, so performing destructive operations (e.g. NCONC) on the value of RECLOOK may leave the record package in an inconsistent state. To change a record declaration, use EDITREC.

(**FIELDLOOK** *FIELDNAME*)   [Function]

Returns the list of declarations in which *FIELDNAME* is the name of a field.

(**RECORDFIELDNAMES** *RECORDNAME —*)   [Function]

Returns the list of fields declared in record *RECORDNAME*. *RECORDNAME* may either be a name or an entire declaration.

(**RECORDACCESS** *FIELD DATUM DEC TYPE NEWVALUE*) [Function]

> *TYPE* is one of FETCH, REPLACE, FFETCH, FREPLACE, /REPLACE or their lowercase equivalents. *TYPE*=NIL means FETCH. If *TYPE* corresponds to a fetch operation, i.e. is FETCH, or FFETCH, RECORDACCESS performs (*TYPE FIELD* of *DATUM*). If *TYPE* corresponds to a replace, RECORDACCESS performs (*TYPE FIELD* of *DATUM* with *NEWVALUE*). *DEC* is an optional declaration; if given, *FIELD* is interpreted as a field name of that declaration.

> Note that RECORDACCESS is relatively inefficient, although it is better than constructing the equivalent form and performing an EVAL.

(**RECORDACCESSFORM** *FIELD DATUM TYPE NEWVALUE*) [Function]

> Returns the form that would be compiled as a result of a record access. *TYPE* is one of FETCH, REPLACE, FFETCH, FREPLACE, /REPLACE or their lowercase equivalents. *TYPE*=NIL means FETCH.

Changetran

Often, you'll want to assign a new value to some datum that is a function of its current value:

Incrementing a counter: (SETQ X (IPLUS X 1))

Pushing an item on the front of a list: (SETQ X (CONS Y X))

Popping an item off a list: (PROG1 (CAR X) (SETQ X (CDR X)))

Those are simple when you're working with a variable; it gets complicated when you're working with structured data. For example, if you want to modify (CAR X), the above examples would be:

```
(CAR (RPLACA X (IPLUS (CAR X) 1)))
(CAR (RPLACA X (CONS Y (CAR X)))
(PROG1 (CAAR X) (RPLACA X (CDAR X)))
```

and if you're changing an element in an array, (ELT A N), the examples would be:

```
(SETA A N (IPLUS (ELT A N) 1)))
(SETA A N (CONS Y (ELT A N))))
(PROG1 (CAR (ELT A N)) (SETA A N (CDR (ELT A N))))
```

Changetran is designed to provide a simpler way to express these common (but user-extensible) structure modifications. Changetran defines a set of CLISP words that encode the kind of modification to take place—pushing on a list, adding to a number, etc. More important, you only indicate the item to be modified once. Thus, the "change word" ADD is used to increase the value of a datum by the sum of a set of numbers. Its arguments are the datum, and a set of numbers to be added to it. The datum must be a variable or an accessing expression (envolving FETCH, CAR, LAST, ELT, etc) that can be translated to the appropriate setting expression.

For example, (ADD X 1) is equivalent to:

```
(SETQ X (PLUS X 1))
```

and (ADD (CADDR X) (FOO)) is equivalent to:

```
(CAR (RPLACA (CDDR X) (PLUS (FOO) (CADDR X))))
```

If the datum is a complicated form involving function calls, such as (ELT (FOO X) (FIE Y))), Changetran goes to some lengths to make sure that those subsidiary functions are evaluated only once, even though they are used in both the setting and accessing parts of the translation. You can rely on the fact that the forms will be evaluated only as often as they appear in your expression.

For ADD and all other changewords, the lowercase version (add, etc.) may also be specified. Like other CLISP words, change words are translated using all CLISP declarations in effect (see Chapter 21).

The following is a list of those change words recognized by Changetran. Except for POP, the value of all built-in changeword forms is defined to be the new value of the datum.

(**ADD** *DATUM ITEM$_1$ ITEM$_2$ ...*) [Change Word]

> Adds the specified items to the current value of the datum, stores the result back in the datum location. The translation will use IPLUS, PLUS, or FPLUS according to the CLISP declarations in effect (see Chapter 21).

(**PUSH** *DATUM ITEM$_1$ ITEM$_2$ ...*) [Change Word]

> CONSes the items onto the front of the current value of the datum, and stores the result back in the datum location. For example, (PUSH X A B) would translate as (SETQ X (CONS A (CONS B X))).

(**PUSHNEW** *DATUM ITEM*) [Change Word]

> Like PUSH (with only one item) except that the item is not added if it is already FMEMB of the datum's value.
>
> Note that, whereas (CAR (PUSH X 'FOO)) will always be FOO, (CAR (PUSHNEW X 'FOO)) might be something else if FOO already existed in the middle of the list.

(**PUSHLIST** *DATUM ITEM$_1$ ITEM$_2$ ...*) [Change Word]

> Similar to PUSH, except that the items are APPENDed in front of the current value of the datum. For example, (PUSHLIST X A B) translates as (SETQ X (APPEND A B X)).

(**POP** *DATUM*) [Change Word]

> Returns CAR of the current value of the datum after storing its CDR into the datum. The current value is computed only once even though it is referenced twice. Note that this is the only built-in changeword for which the value of the form is not the new value of the datum.

(**SWAP** *DATUM$_1$ DATUM$_2$*) [Change Word]

> Sets *DATUM$_1$* to *DATUM$_2$* and vice versa.

(**CHANGE** *DATUM FORM*) [Change Word]

> This is the most flexible of all change words: You give an arbitrary form describing what the new value should be. But it still highlights the fact that structure modification is happening, and still lets the datum appear only once. CHANGE sets *DATUM* to the value of *FORM**, where *FORM** is constructed from *FORM* by substituting the datum expression for every occurrence of the symbol DATUM. For example,
>
>   (CHANGE (CAR X) (ITIMES DATUM 5))
>
> translates as

```
(CAR (RPLACA X (ITIMES (CAR X) 5))).
```

CHANGE is useful for expressing modifications that are not built-in and are not common enough to justify defining a user-changeword.

You can define new change words. To define a change word, say sub, that subtracts items from the current value of the datum, you must put the property CLISPWORD, value (CHANGETRAN . sub) on both the upper- and lower-case versions of sub:

```
(PUTPROP 'SUB 'CLISPWORD '(CHANGETRAN . sub))
(PUTPROP 'sub 'CLISPWORD '(CHANGETRAN . sub))
```

Then, you must put (on the *lower*-case version of sub only) the property CHANGEWORD, with value *FN*. *FN* is a function that will be applied to a single argument, the whole sub form, and must return a form that Changetran can translate into an appropriate expression. This form should be a list structure with the symbol DATUM used whenever you want an accessing expression for the current value of the datum to appear. The form (DATUM← *FORM*) (note that DATUM← is a single symbol) should occur once in the expression; this specifies that an appropriate storing expression into the datum should occur at that point. For example, sub could be defined as:

```
(PUTPROP 'sub 'CHANGEWORD
        '(LAMBDA (FORM)
            (LIST 'DATUM←
                (LIST 'IDIFFERENCE
                     'DATUM
                     (CONS 'IPLUS (CDDR FORM))))))
```

If the expression (sub (CAR X) A B) were encountered, the arguments to SUB would first be dwimified, and then the CHANGEWORD function would be passed the list (sub (CAR X) A B), and return (DATUM← (IDIFFERENCE DATUM (IPLUS A B))), which Changetran would convert to (CAR (RPLACA X (IDIFFERENCE (CAR X) (IPLUS A B)))).

Note: The sub changeword as defined above will always use IDIFFERENCE and IPLUS; add uses the correct addition operation depending on the current CLISP declarations (see Chapter 21).

Built-In and User Data Types

Medley is a system for manipulating various kinds of data; it comes with a large set of built-in data types, which you can use to represent a variety of abstract objects; you can also define additional "user data types" that you can manipulate exactly like built-in data types.

Each data type in Medley has an associated "type name," a symbol. Some of the type names of built-in data types are: LITATOM, LISTP, STRINGP, ARRAYP, STACKP, SMALLP, FIXP, and FLOATP. For user data types, the type name is specified when the data type is created.

(**DATATYPES** —)   [Function]

Returns a list of all type names currently defined.

(**USERDATATYPES**)   [Function]

Returns list of names of currently declared user data types.

(**TYPENAME** *DATUM*)   [Function]

Returns the type name for the data type of *DATUM*.

(**TYPENAMEP** *DATUM TYPE*)   [Function]

>Returns ᴛ if *DATUM* is an object with type name equal to *TYPE*, otherwise ɴɪʟ.

In addition to built-in data-types like symbols, lists, arrays, etc., Medley provides a way to define completely *new* classes of objects, with a fixed number of fields determined by the definition of the data type. To define a new class of objects, you must supply a name for the new data type and specifications for each of its fields. Each field may contain either a pointer (i.e., any arbitrary Interlisp datum), an integer, a floating point number, or an *N*-bit integer.

Note: The most convenient way to define new user data types is via ᴅᴀᴛᴀᴛʏᴘᴇ record declarations (see Chapter 8) which call the following functions.

(**DECLAREDATATYPE** *TYPENAME FIELDSPECS* — —)   [Function]

>Defines a new user data type, with the name *TYPENAME*. *FIELDSPECS* is a list of "field specifications." Each field specification may be one of the following:

>>ᴘᴏɪɴᴛᴇʀ   Field may contain any Interlisp datum.

>>ғɪxᴘ   Field contains an integer.

>>ғʟᴏᴀᴛᴘ   Field contains a floating point number.

>>(ʙɪᴛs *N*)   Field contains a non-negative integer less than 2^N.

>>ʙʏᴛᴇ   Equivalent to (ʙɪᴛs 8).

>>ᴡᴏʀᴅ   Equivalent to (ʙɪᴛs 16).

>>sɪɢɴᴇᴅᴡᴏʀᴅ   Field contains a 16 bit signed integer.

>>>ᴅᴇᴄʟᴀʀᴇᴅᴀᴛᴀᴛʏᴘᴇ returns a list of "field descriptors," one for each element of *FIELDSPECS*. A field descriptor contains information about where within the datum the field is actually stored.

>>>If *FIELDSPECS* is ɴɪʟ, *TYPENAME* is "undeclared." If *TYPENAME* is already declared as a data type, it is undeclared, and then re-declared with the new *FIELDSPECS*. An instance of a data type that has been undeclared has a type name of **ᴅᴇᴀʟʟᴏᴄ**.

(**FETCHFIELD** *DESCRIPTOR DATUM*)   [Function]

>Returns the contents of the field described by *DESCRIPTOR* from *DATUM*. *DESCRIPTOR* must be a "field descriptor" as returned by ᴅᴇᴄʟᴀʀᴇᴅᴀᴛᴀᴛʏᴘᴇ or ɢᴇᴛᴅᴇsᴄʀɪᴘᴛᴏʀs. If *DATUM* is not an instance of the datatype of which *DESCRIPTOR* is a descriptor, causes error ᴅᴀᴛᴜᴍ ᴏғ ɪɴᴄᴏʀʀᴇᴄᴛ ᴛʏᴘᴇ.

(**REPLACEFIELD** *DESCRIPTOR DATUM NEWVALUE*)   [Function]

>Store *NEWVALUE* into the field of *DATUM* described by *DESCRIPTOR*. *DESCRIPTOR* must be a field descriptor as returned by ᴅᴇᴄʟᴀʀᴇᴅᴀᴛᴀᴛʏᴘᴇ. If *DATUM* is not an instance of the

datatype of which *DESCRIPTOR* is a descriptor, causes error Datum of incorrect type. Value is *NEWVALUE*.

(**NCREATE** *TYPE OLDOBJ*)   [Function]

Creates and returns a new instance of datatype *TYPE*.

If *OLDOBJ* is also a datum of datatype *TYPE*, the fields of the new object are initialized to the values of the corresponding fields in *OLDOBJ*.

NCREATE will not work for built-in datatypes, such as ARRAYP, STRINGP, etc. If *TYPE* is not the type name of a previously declared *user* data type, generates an error, Illegal data type.

(**GETFIELDSPECS** *TYPENAME*)   [Function]

Returns a list which is EQUAL to the *FIELDSPECS* argument given to DECLAREDATATYPE for *TYPENAME*; if *TYPENAME* is not a currently declared data-type, returns NIL.

(**GETDESCRIPTORS** *TYPENAME*)   [Function]

Returns a list of field descriptors, EQUAL to the *value* of DECLAREDATATYPE for *TYPENAME*. If *TYPENAME* is not an atom, (TYPENAME *TYPENAME*) is used.

You can define how a user data type prints, using DEFPRINT (see Chapter 25), how they are to be evaluated by the interpreter via DEFEVAL (see Chapter 10), and how they are to be compiled by the compiler via COMPILETYPELST (see Chapter 18).

INTERLISP-D REFERENCE MANUAL

[This page intentionally left blank]

Medley gives you a large number of predicates, conditional functions, and control functions. Also, there is a complex "iterative statement" facility which allows you to easily create complex loops and iterative constructs.

Data Type Predicates

Medley provides separate functions for testing whether objects are of certain commonly-used types:

(**LITATOM** *X*) [Function]

    Returns T if *X* is a symbol; NIL otherwise. Note that a number is not a symbol.

(**SMALLP** *X*) [Function]

    Returns *X* if *X* is a small integer; NIL otherwise. (The range of small integers is -65536 to +65535.

(**FIXP** *X*) [Function]

    Returns *X* if *X* is a small or large integer; NIL otherwise.

(**FLOATP** *X*) [Function]

    Returns *X* if *X* is a floating point number; NIL otherwise.

(**NUMBERP** *X*) [Function]

    Returns *X* if *X* is a number of any type, NIL otherwise.

(**ATOM** *X*) [Function]

    Returns T if *X* is an atom (i.e. a symbol or a number); NIL otherwise.

    (ATOM *X*) is NIL if *X* is an array, string, etc. In Common Lisp, CL:ATOM is defined equivalent to the Interlisp function NLISTP.

(**LISTP** *X*) [Function]

    Returns *X* if *X* is a list cell (something created by CONS); NIL otherwise.

(**NLISTP** *X*) [Function]

    (NOT (LISTP X)). Returns T if *X* is not a list cell, NIL otherwise.

(**STRINGP** *X*) [Function]

    Returns *X* if *X* is a string, NIL otherwise.

(**ARRAYP** *X*) [Function]

    Returns *X* if *X* is an array, NIL otherwise.

(**HARRAYP** *X*) [Function]

    Returns *X* if it is a hash array object; otherwise NIL.

HARRAYP returns NIL if *X* is a list whose CAR is an HARRAYP, even though this is accepted by the hash array functions.

Note: The empty list, () or NIL, is considered to be a symbol, rather than a list. Therefore, (LITATOM NIL) = (ATOM NIL) = T and (LISTP NIL) = NIL. Take care when using these functions if the object may be the empty list NIL.

Equality Predicates

Sometimes, there is more than one type of equality. For instance, given two lists, you can ask whether they are exactly the same object, or whether they are two distinct lists that contain the same elements. Confusion between these two types of equality is often the source of program errors.

(**EQ** *X Y*) [Function]

Returns T if *X* and *Y* are identical pointers; NIL otherwise. EQ should not be used to compare two numbers, unless they are small integers; use EQP instead.

(**NEQ** *X Y*) [Function]

The same as (NOT (EQ *X Y*))

(**NULL** *X*) [Function]
(**NOT** *X*) [Function]

The same as (EQ *X* NIL)

(**EQP** *X Y*) [Function]

Returns T if *X* and *Y* are EQ, or if *X* and *Y* are numbers and are equal in value; NIL otherwise. For more discussion of EQP and other number functions, see Chapter 7.

EQP also can be used to compare stack pointers (Section 11) and compiled code (Chapter 10).

(**EQUAL** *X Y*) [Function]

EQUAL returns T if *X* and *Y* are one of the following:

1. EQ
2. EQP, i.e., numbers with equal value
3. STREQUAL, i.e., strings containing the same sequence of characters
4. Lists and CAR of *X* is EQUAL to CAR of *Y*, and CDR of *X* is EQUAL to CDR of *Y*

EQUAL returns NIL otherwise. Note that EQUAL can be significantly slower than EQ.

A loose description of EQUAL might be to say that *X* and *Y* are EQUAL if they print out the same way.

(**EQUALALL** *X Y*) [Function]

Like EQUAL, except it descends into the contents of arrays, hash arrays, user data types, etc. Two non-EQ arrays may be EQUALALL if their respective componants are EQUALALL.

Note: In general, EQUALALL descends all the way into all datatypes, both those you've defined and those built into the system. If you have a data structure with fonts and pointers to windows, EQUALALL will descend those also. If the data structures are circular, as windows are, EQUALALL can cause stack overflow.

Logical Predicates

(**AND** X_1 X_2 ... X_N)   [NLambda NoSpread Function]

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument evaluates to NIL, AND immediately returns NIL, without evaluating the remaining arguments. If all of the arguments evaluate to non-NIL, the value of the last argument is returned. (AND) => T.

(**OR** X_1 X_2 ... X_N)   [NLambda NoSpread Function]

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument is non-NIL, the value of that argument is returned by OR (without evaluating the remaining arguments). If all of the arguments evaluate to NIL, NIL is returned. (OR) => NIL.

AND and OR can be used as simple logical connectives, but note that they may not evaluate all of their arguments. This makes a difference if some of the arguments cause side-effects. This also means you can use AND and OR as simple conditional statements. For example: (AND (LISTP X) (CDR X)) returns the value of (CDR X) if X is a list cell; otherwise it returns NIL without evaluating (CDR X). In general, you should avoid this use of AND and OR in favor of more explicit conditional statements in order to make programs more readable.

COND Conditional Function

(**COND** $CLAUSE_1$ $CLAUSE_2$ ... $CLAUSE_K$)   [NLambda NoSpread Function]

COND takes an indefinite number of arguments, called clauses. Each $CLAUSE_i$ is a list of the form (P_i C_{i1} ... C_{iN}), where P_i is the predicate, and C_{i1} ... C_{iN} are the consequents. The operation of COND can be paraphrased as:

IF P_1 THEN C_{11} ... C_{1N} ELSEIF P_2 THEN C_{21} ... C_{2N} ELSEIF P_3 ...

The clauses are considered in sequence as follows: The predicate P_1 of the clause $CLAUSE_i$ is evaluated. If the value of P_1 is "true" (non-NIL), the consequents C_{i1} ... C_{iN} are evaluated in order, and the value of the COND is the value of the last expression in the clause. If P_1 is "false" (EQ to NIL), then the remainder of $CLAUSE_i$ is ignored, and the next clause, $CLAUSE_{i+1}$, is considered. If no P_i is true for *any* clause, the value of the COND is NIL.

If a clause has no consequents, and has the form (P_i), then if P_i evaluates to non-NIL, it is returned as the value of the COND. It is only evaluated once.

Example:

```
←(DEFINEQ (DOUBLE (X)
    (COND ((NUMBERP X) (PLUS X X))
```

```
                         ((STRINGP X) (CONCAT X X))
                         ((ATOM X) (PACK* X X))
                         (T (PRINT "unknown") X)
                         ((HORRIBLE-ERROR))]
                   (DOUBLE)
             ←(DOUBLE 5)
                   10

             ←(DOUBLE "FOO")
                   "FOOFOO"

             ←(DOUBLE 'BAR)
                   BARBAR

             ←(DOUBLE '(A B C))
                   "unknown"
                   (A B C)
```

A few points about this example: Notice that 5 is both a number and an atom, but it is "caught" by the NUMBERP clause before the ATOM clause. Also notice the predicate T, which is always true. This is the normal way to indicate a COND clause which will always be executed (if none of the preceeding clauses are true). (HORRIBLE-ERROR) will never be executed.

The IF Statement

The IF statement lets you write conditional expressions that are easier to read than using COND directly. CLISP translates expressions using IF, THEN, ELSEIF, or ELSE (or their lowercase versions) into equivalent CONDs. In general, statements of the form:

```
      (if AAA then BBB elseif CCC then DDD else EEE)
```

are translated to:

```
      (COND (AAA BBB)
            (CCC DDD)
            (T EEE))
```

The segment between IF or ELSEIF and the next THEN corresponds to the predicate of a COND clause, and the segment between THEN and the next ELSE or ELSEIF as the consequent(s). ELSE is the same as ELSEIF T THEN. These words are spelling corrected using the spelling list CLISPIFWORDSPLST. You may also use lower-case versions (if, then, elseif, else).

If there is nothing following a THEN, or THEN is omitted entirely, the resulting COND clause has a predicate but no consequent. For example, (if X then elseif ...) and (if X elseif ...) both translate to (COND (X) ...)—if X is not NIL, it is returned as the value of the COND.

Each predicate must be a single expression, but multiple expressions are allowed as the consequents after THEN or ELSE. Multiple consequent expressions are implicitly wrapped in a PROGN, and the value of the last one is returned as the value of the consequent. For example:

```
(if X then (PRINT "FOO") (PRINT "BAR") elseif Y then (PRINT "BAZ"))
```

Selection Functions

(**SELECTQ** X $CLAUSE_1$ $CLAUSE_2$ ... $CLAUSE_K$
$DEFAULT$)   [NLambda NoSpread Function]

Selects a form or sequence of forms based on the value of X. Each clause $CLAUSE_i$ is a list
of the form (S_i C_{i1} ... C_{iN}) where S_i is the selection key. Think of SELECTQ as:

```
IF X = S₁ THEN C₁₁ ... C₁ₙ ELSEIF X = S₂
        THEN ... ELSE DEFAULT
```

If S_i is a symbol, the value of X is tested to see if it is EQ to S_i (which is *not* evaluated). If
so, the expressions C_{i1} ... C_{iN} are evaluated in sequence, and the value of the SELECTQ
is the value of the last expression.

If S_i is a list, the value of X is compared with each element (not evaluated) of S_i, and if X is
EQ to any one of them, then C_{i1} ... C_{iN} are evaluated as above.

If $CLAUSE_i$ is not selected in one of the two ways described, $CLAUSE_{i+1}$ is tested, etc., until
all the clauses have been tested. If none is selected, $DEFAULT$ is evaluated, and its value is
returned as the value of the SELECTQ. $DEFAULT$ must be present.

An example of the form of a SELECTQ is:

```
[SELECTQ MONTH
    (FEBRUARY (if (LEAPYEARP) then 29 else 28))
        ((SEPTEMBER APRIL JUNE NOVEMBER) 30) 31]
```

If the value of MONTH is the symbol FEBRUARY, the SELECTQ returns 28 or 29 (depending
on (LEAPYEARP)); otherwise if MONTH is APRIL, JUNE, SEPTEMBER, or NOVEMBER, the
SELECTQ returns 30; otherwise it returns 31.

SELECTQ compiles open, and is therefore very fast; however, it will not work if the value
of X is a list, a large integer, or floating point number, since SELECTQ uses EQ for all
comparisons.

SELCHARQ (Chapter 2) is a version of SELECTQ that recognizes CHARCODE symbols.

(**SELECTC** X $CLAUSE_1$ $CLAUSE_2$ ... $CLAUSE_K$
$DEFAULT$)   [NLambda NoSpread Function]

"SELECTQ-on-Constant." Like SELECTQ, but the selection keys are evaluated, and the
result used as a SELECTQ-style selection key.

SELECTC is compiled as a SELECTQ, with the selection keys evaluated at compile-time.
Therefore, the selection keys act like compile-time constants (see Chapter 18).

For example:

```
[SELECTC NUM
  ((for X from 1 to 9 collect (TIMES X X)) "SQUARE") "HIP"]
```

compiles as:

```
(SELECTQ NUM
  ((1 4 9 16 25 36 49 64 81) "SQUARE") "HIP")
```

PROG and Associated Control Functions

(**PROG1** $X_1 X_2 \ldots X_N$)   [NLambda NoSpread Function]

> Evaluates its arguments in order, and returns the value of its first argument X_1. For example, (PROG1 X (SETQ X Y)) sets X to Y, and returns X's original value.

(**PROG2** $X_1 X_2 \ldots X_N$)   [NoSpread Function]

> Like PROG1. Evaluates its arguments in order, and returns the value of its second argument X_2.

(**PROGN** $X_1 X_2 \ldots X_N$)   [NLambda NoSpread Function]

> PROGN evaluates each of its arguments in order, and returns the value of its last argument. PROGN is used to specify more than one computation where the syntax allows only one, e.g., (SELECTQ ... (PROGN ...)) allows evaluation of several expressions as the default condition for a SELECTQ.

(**PROG** $VARLST E_1 E_2 \ldots E_N$)   [NLambda NoSpread Function]

> Lets you bind some variables while you execute a series of expressions. *VARLST* is a list of local variables (must be NIL if no variables are used). Each symbol in *VARLST* is treated as the name of a local variable and bound to NIL. *VARLST* can also contain lists of the form (*NAME FORM*). In this case, *NAME* is the name of the variable and is bound to the value of *FORM*. The evaluation takes place before any of the bindings are performed, e.g., (PROG ((X Y) (Y X)) ...) will bind local variable X to the value of Y (evaluated *outside* the PROG) and local variable Y to the value of X (outside the PROG). An attempt to use anything other than a symbol as a PROG variable will cause an error, Arg not symbol. An attempt to use NIL or T as a PROG variable will cause an error, Attempt to bind NIL or T.
>
> The rest of the PROG is a sequence of forms and symbols (labels). The forms are evaluated sequentially; the labels serve only as markers. The two special functions, GO and RETURN, alter this flow of control as described below. The value of the PROG is usually specified by the function RETURN. If no RETURN is executed before the PROG "falls off the end," the value of the PROG is NIL.

(**GO** L)   [NLambda NoSpread Function]

> GO is used to cause a transfer in a PROG. (GO L) will cause the PROG to evaluate forms starting at the label L (GO does not evaluate its argument). A GO can be used at any level in a PROG. If the label is not found, GO will search higher progs *within the same function,* e.g., (PROG ... A ... (PROG ... (GO A))). If the label is not found in the function in which the PROG appears, an error is generated, Undefined or illegal GO.

(**RETURN** X)   [Function]

> A RETURN is the normal exit for a PROG. Its argument is evaluated and is immediately returned the value of the PROG in which it appears.

Note: If a GO or RETURN is executed in an interpreted function which is not a PROG, the GO or RETURN will be executed in the last interpreted PROG entered if any, otherwise cause an error.

GO or RETURN inside of a compiled function that is not a PROG is not allowed, and will cause an error at compile time.

As a corollary, GO or RETURN in a functional argument, e.g., to SORT, will not work compiled. Also, since NLSETQ's and ERSETQ's compile as *separate* functions, a GO or RETURN *cannot* be used inside of a compiled NLSETQ or ERSETQ if the corresponding PROG is outside, i.e., above, the NLSETQ or ERSETQ.

(**LET** *VARLST $E_1 E_2 \ldots E_N$*)   [Macro]

LET is essentially a PROG that can't contain GO's or RETURN's, and whose last form is the returned value.

(**LET*** *VARLST $E_1 E_2 \ldots E_N$*)   [Macro]
(**PROG*** *VARLST $E_1 E_2 \ldots E_N$*)   [Macro]

LET* and PROG* differ from LET and PROG only in that the binding of the bound variables is done "sequentially." Thus

```
(LET* ((A (LIST 5))
       (B (LIST A A)))
      (EQ A (CADR B)))
```

would evaluate to T; whereas the same form with LET might find A an unbound variable when evaluating (LIST A A).

The Iterative Statement

The various forms of the iterative statement (i.s.) let you write complex loops easily. Rather than writing PROG, MAPC, MAPCAR, etc., let Medley do it for you.

An iterative statement is a form consisting of a number of special words (known as i.s. operators or i.s.oprs), followed by operands. Many i.s.oprs (FOR, DO, WHILE, etc.) act like loops in other programming languages; others (COLLECT, JOIN, IN, etc.) do things useful in Lisp. You can also use lower-case versions of i.s.oprs (do, collect, etc.).

```
← (for X from 1 to 5 do (PRINT 'FOO))
FOO
FOO
FOO
FOO
FOO
NIL
←(for X from 2 to 10 by 2 collect (TIMES X X))
(4 16 36 64 100)
←(for X in '(A B 1 C 6.5 NIL (45)) count (NUMBERP X))
2
```

Iterative statements are implemented using CLISP, which translates them into the appropriate PROGs, MAPCARs, etc. They're are translated using all CLISP declarations in effect (standard/fast/undoable/

etc.); see Chapter 21. Misspelled i.s.oprs are recognized and corrected using the spelling list `CLISPFORWORDSPLST`. Operators can appear in any order; CLISP scans the entire statement before it begins to translate.

If you define a function with the same name as an i.s.opr (`WHILE`, `TO`, etc.), that i.s.opr will no longer cause looping when it appears as `CAR` of a form, although it will continue to be treated as an i.s.opr if it appears in the interior of an iterative statement. To alert you, a warning message is printed, e.g., (`While defined, therefore disabled in CLISP`).

I.S. Types

Every iterative statement must have exactly one of the following operators in it (its "is.stype"), to specify what happens on each iteration. Its operand is called the "body" of the iterative statement.

DO *FORMS*   [I.S. Operator]

> Evaluate *FORMS* at each iteration. `DO` with no other operator specifies an infinite loop. If some explicit or implicit terminating condition is specified, the value of the loop is `NIL`. Translates to `MAPC` or `MAP` whenever possible.

COLLECT *FORM*   [I.S. Operator]

> The value of *FORM* at each iteration is collected in a list, which is returned as the value of the loop when it terminates. Translates to `MAPCAR`, `MAPLIST` or `SUBSET` whenever possible.

> When `COLLECT` translates to a `PROG` (if `UNTIL`, `WHILE`, etc. appear in the loop), the translation employs an open `TCONC` using two pointers similar to that used by the compiler for compiling `MAPCAR`. To disable this translation, perform (`CLDISABLE 'FCOLLECT`).

JOIN *FORM*   [I.S. Operator]

> *FORM* returns a list; the lists from each iteration are concatenated using `NCONC`, forming one long list. Translates to `MAPCONC` or `MAPCON` whenever possible. `/NCONC`, `/MAPCONC`, and `/MAPCON` are used when the CLISP declaration `UNDOABLE` is in effect.

SUM *FORM*   [I.S. Operator]

> The values of *FORM* from each iteration are added together and returned as the value of the loop, e.g., (`for I from 1 to 5 sum (TIMES I I)`) returns `1+4+9+16+25 = 55`. `IPLUS`, `FPLUS`, or `PLUS` will be used in the translation depending on the CLISP declarations in effect.

COUNT *FORM*   [I.S. Operator]

> Counts the number of times that *FORM* is true, and returns that count as the loop's value.

ALWAYS *FORM*   [I.S. Operator]

> Returns `T` if the value of *FORM* is non-`NIL` for all iterations. **Note**: Returns `NIL` as soon as the value of *FORM* is `NIL`).

NEVER *FORM*   [I.S. Operator]

> Like ALWAYS, but returns T if the value of *FORM* is *never* true. **Note**: Returns NIL as soon as the value of *FORM* is non-NIL.

Often, you'll want to set a variable each time through the loop; that's called the "iteration variable", or i.v. for short. The following i.s.types explicitly refer to the i.v. This is explained below under FOR.

THEREIS *FORM*   [I.S. Operator]

> Returns the first value of the i.v. for which *FORM* is non-NIL, e.g., (for X in Y thereis (NUMBERP X)) returns the first number in Y.

> **Note**: Returns the value of the i.v. as soon as the value of *FORM* is non-NIL.

LARGEST *FORM*   [I.S. Operator]
SMALLEST *FORM*   [I.S. Operator]

> Returns the value of the i.v. that provides the largest/smallest value of *FORM*. $$EXTREME is always bound to the current greatest/smallest value, $$VAL to the value of the i.v. from which it came.

Iteration Variable I.s.oprs

You'll want to bind variables to use during the loop. Rather than putting the loop inside a PROG or LET, you can specify bindings like so:

BIND *VAR*   [I.S. Operator]
BIND *VARS*   [I.S. Operator]

> Used to specify dummy variables, which are bound locally within the i.s.

> **Note**:   You can initialize a variable *VAR* by saying *VAR*←*FORM*:

> > (bind HEIGHT ← 0 WEIGHT ← 0 for SOLDIER in ...)

To specify iteration variables, use these operators:

FOR *VAR*   [I.S. Operator]

> Specifies the iteration variable (i.v.) that is used in conjunction with IN, ON, FROM, TO, and BY. The variable is rebound within the loop, so the value of the variable outside the loop is not affected. Example:

> > ```
> > ←(SETQ X 55)
> > 55
> > ←(for X from 1 to 5 collect (TIMES X X))
> > (1 4 9 16 25)
> > ←X
> > 55
> > ```

FOR OLD *VAR*   [I.S. Operator]

> Like FOR, but *VAR* is *not* rebound, so its value outside the loop *is* changed.   Example:

> > ```
> > ←(SETQ X 55)
> > 55
> > ```

```
←(for old X from 1 to 5 collect (TIMES X X))
(1 4 9 16 25)
←X
6
```

FOR *VARS*   [I.S. Operator]

> *VARS* a list of variables, e.g., (for (X Y Z) in ...). The first variable is the i.v., the
> rest are dummy variables. See BIND above.

IN *FORM*   [I.S. Operator]

> *FORM* must evaluate to a list. The i.v. is set to successive elements of the list, one per
> iteration. For example, (for X in Y do ...) corresponds to (MAPC Y (FUNCTION
> (LAMBDA (X) ...))). If no i.v. has been specified, a dummy is supplied, e.g., (in Y
> collect CADR) is equivalent to (MAPCAR Y (FUNCTION CADR)).

ON *FORM*   [I.S. Operator]

> Same as IN, but the i.v. is reset to the corresponding *tail* at each iteration. Thus IN
> corresponds to MAPC, MAPCAR, and MAPCONC, while ON corresponds to MAP, MAPLIST,
> and MAPCON.
>
> ```
> ←(for X on '(A B C) do (PRINT X))
> (A B C)
> (B C)
> (C)
> NIL
> ```
>
> **Note**: For both IN and ON, *FORM* is evaluated before the main part of the i.s. is entered, i.e.
> *outside* of the scope of any of the bound variables of the i.s. For example, (for X bind
> (Y←'(1 2 3)) in Y ...) will map down the list which is the value of Y evaluated
> *outside* of the i.s., *not* (1 2 3).

IN OLD *VAR*   [I.S. Operator]

> Specifies that the i.s. is to iterate down *VAR*, with *VAR* itself being reset to the
> corresponding tail at each iteration, e.g., after (for X in old L do ... until
> ...) finishes, L will be some tail of its original value.

IN OLD (*VAR←FORM*)   [I.S. Operator]

> Same as IN OLD *VAR*, except *VAR* is first set to value of *FORM*.

ON OLD *VAR*   [I.S. Operator]

> Same as IN OLD *VAR* except the i.v. is reset to the current value of *VAR* at each iteration,
> instead of to (CAR *VAR*).

ON OLD (*VAR←FORM*)   [I.S. Operator]

> Same as ON OLD *VAR*, except *VAR* is first set to value of *FORM*.

INSIDE *FORM*   [I.S. Operator]

Like IN, but treats first non-list, non-NIL tail as the last element of the iteration, e.g., INSIDE '(A B C D . E) iterates five times with the i.v. set to E on the last iteration. INSIDE 'A is equivalent to INSIDE '(A), which will iterate once.

FROM *FORM*   [I.S. Operator]

Specifies the initial value for a numerical i.v. The i.v. is automatically incremented by 1 after each iteration (unless BY is specified). If no i.v. has been specified, a dummy i.v. is supplied and initialized, e.g., (from 2 to 5 collect SQRT) returns (1.414 1.732 2.0 2.236).

TO *FORM*   [I.S. Operator]

Specifies the final value for a numerical i.v. If FROM is not specified, the i.v. is initialized to 1. If no i.v. has been specified, a dummy i.v. is supplied and initialized. If BY is not specified, the i.v. is automatically incremented by 1 after each iteration. When the i.v. is definitely being *incremented*, i.e., either BY is not specified, or its operand is a positive number, the i.s. terminates when the i.v. exceeds the value of *FORM*. Similarly, when the i.v. is definitely being decremented the i.s. terminates when the i.v. becomes *less* than the value of *FORM* (see description of BY).

FORM is evaluated only once, when the i.s. is first entered, and its value bound to a temporary variable against which the i.v. is checked each iteration. If the user wishes to specify an i.s. in which the value of the boundary condition is recomputed each iteration, he should use WHILE or UNTIL instead of TO.

When both the operands to TO and FROM are numbers, and TO's operand is less than FROM's operand, the i.v. is decremented by 1 after each iteration. In this case, the i.s. terminates when the i.v. becomes *less* than the value of *FORM*. For example, (from 10 to 1 do PRINT) prints the numbers from 10 down to 1.

BY *FORM* (without IN or ON)   [I.S. Operator]

If you aren't using IN or ON, BY specifies how the i.v. itself is reset at each iteration. If you're using FROM or TO, the i.v. is known to be numerical, so the new i.v. is computed by adding the value of *FORM* (which is reevaluated each iteration) to the current value of the i.v., e.g., (for N from 1 to 10 by 2 collect N) makes a list of the first five odd numbers.

If *FORM* is a positive number (*FORM* itself, not its value, which in general CLISP would have no way of knowing in advance), the loop stops when the value of the i.v. *exceeds* the value of TO's operand. If *FORM* is a negative number, the loop stops when the value of the i.v. becomes *less* than TO's operand, e.g., (for I from N to M by -2 until (LESSP I M) ...). Otherwise, the terminating condition for each iteration depends on the value of *FORM* for that iteration: if *FORM*<0, the test is whether the i.v. is less than TO's operand, if *FORM*>0 the test is whether the i.v. exceeds TO's operand; if *FORM* = 0, the loop terminates unconditionally.

If you didn't use FROM or TO and *FORM* is not a number, the i.v. is simply reset to the value of *FORM* after each iteration, e.g., (for I from N by (FOO) ...) sets I to the value of (FOO) on each loop after the first.

BY *FORM* (with IN or ON)   [I.S. Operator]

If you did use IN or ON, *FORM*'s value determines the *tail* for the next iteration, which in turn determines the value for the i.v. as described earlier, i.e., the new i.v. is CAR of the tail for IN, the tail itself for ON. In conjunction with IN, you can refer to the current tail within *FORM* by using the i.v. or the operand for IN/ON, e.g., (for Z in L by (CDDR Z) ...) or (for Z in L by (CDDR L) ...). At translation time, the name of the internal variable which holds the value of the current tail is substituted for the i.v. throughout *FORM*. For example, (for X in Y by (CDR (MEMB 'FOO (CDR X))) collect X) specifies that after each iteration, CDR of the current tail is to be searched for the atom FOO, and (CDR of) this latter tail to be used for the next iteration.

AS *VAR*   [I.S. Operator]

Lets you have more than one i.v. for a single loop, e.g., (for X in Y as U in V do ...) moves through the lisps Y and V in parallel (see MAP2C). The loop ends when any of the terminating conditions is met, e.g., (for X in Y as I from 1 to 10 collect X) makes a list of the first ten elements of Y, or however many elements there are on Y if less than 10.

The operand to AS, *VAR*, specifies the new i.v. For the remainder of the i.s., or until another AS is encountered, all operators refer to the new i.v. For example, (for I from 1 to N_1 as J from 1 to N_2 by 2 as K from N_3 to 1 by -1 ...) terminates when I exceeds N_1, or J exceeds N_2, or K becomes less than 1. After each iteration, I is incremented by 1, J by 2, and K by -1.

OUTOF *FORM*   [I.S. Operator]

For use with generators. On each iteration, the i.v. is set to successive values returned by the generator. The loop ends when the generator runs out.

Condition I.S. Oprs

What if you want to do things only on certain times through the loop? You could make the loop body a big COND, but it's much more readable to use one of these:

WHEN *FORM*   [I.S. Operator]

Only run the loop body when *FORM*'s value is non-NIL. For example, (for X in Y collect X when (NUMBERP X)) collects only the elements of Y that are numbers.

UNLESS *FORM*   [I.S. Operator]

Opposite of WHEN: WHEN Z is the same as UNLESS (NOT Z).

WHILE *FORM*   [I.S. Operator]

WHILE *FORM* evaluates *FORM* *before* each iteration, and if the value is NIL, exits.

UNTIL *FORM* [I.S. Operator]

Opposite of WHILE: Evaluates *FORM before* each iteration, and if the value is *not* NIL, exits.

REPEATWHILE *FORM* [I.S. Operator]

Same as WHILE except the test is performed *after* the loop body, but before the i.v. is reset for the next iteration.

REPEATUNTIL *FORM* [I.S. Operator]

Same as UNTIL, except the test is performed *after* the loop body.

Other I.S. Operators

FIRST *FORM* [I.S. Operator]

FORM is evaluated once before the first iteration, e.g., (for X Y Z in L first (FOO Y Z) ...), and FOO could be used to initialize Y and Z.

FINALLY *FORM* [I.S. Operator]

FORM is evaluated after the loop terminates. For example, (for X in L bind Y_0 do (if (ATOM X) then (SETQ Y (PLUS Y 1))) finally (RETURN Y)) will return the number of atoms in L.

EACHTIME *FORM* [I.S. Operator]

FORM is evaluated at the beginning of each iteration before, and regardless of, any testing. For example, consider,

```
(for I from 1 to N
    do (... (FOO I) ...)
    unless (... (FOO I) ...)
    until (... (FOO I) ...))
```

You might want to set a temporary variable to the value of (FOO I) in order to avoid computing it three times each iteration. However, without knowing the translation, you can't know whether to put the assignment in the operand to DO, UNLESS, or UNTIL. You can avoid this problem by simply writing EACHTIME (SETQ J (FOO I)).

DECLARE: *DECL* [I.S. Operator]

Inserts the form (DECLARE *DECL*) immediately following the PROG variable list in the translation, or, in the case that the translation is a mapping function rather than a PROG, immediately following the argument list of the lambda expression in the translation. This can be used to declare variables bound in the iterative statement to be compiled as local or special variables. For example (for X in Y declare: (LOCALVARS X) ...). Several DECLARE:s can apppear in the same i.s.; the declarations are inserted in the order they appear.

DECLARE *DECL* [I.S. Operator]

Same as DECLARE:.

Since `DECLARE` is also the name of a function, `DECLARE` cannot be used as an i.s. operator when it appears as `CAR` of a form, i.e. as the first i.s. operator in an iterative statement. However, `declare` (lowercase version) *can* be the first i.s. operator.

ORIGINAL *I.S.OPR OPERAND*   [I.S. Operator]

I.S.OPR will be translated using its original, built-in interpretation, independent of any user defined i.s. operators.

There are also a number of i.s.oprs that make it easier to create iterative statements that use the clock, looping for a given period of time. See timers, Chapter 12.

Miscellaneous Hints For Using I.S.Oprs

Lowercase versions of all i.s. operators are equivalent to the uppercase, e.g., `(for X in Y ...)` is equivalent to `(FOR X IN Y ...)`.

Each i.s. operator is of lower precedence than all Interlisp forms, so parentheses around the operands can be omitted, and will be supplied where necessary, e.g., `BIND (X Y Z)` can be written `BIND X Y Z`, `OLD (X_FORM)` as `OLD X_FORM`, etc.

`RETURN` or `GO` may be used in any operand. (In this case, the translation of the iterative statement will always be in the form of a `PROG`, never a mapping function.) `RETURN` means return from the loop (with the indicated value), *not* from the function in which the loop appears. `GO` refers to a label elsewhere in the function in which the loop. appears, except for the labels `$$LP`, `$$ITERATE`, and `$$OUT` which are reserved, as described below.

In the case of `FIRST`, `FINALLY`, `EACHTIME`, `DECLARE:` or one of the i.s.types, e.g., `DO`, `COLLECT`, `SUM`, etc., the operand can consist of more than one form, e.g., `COLLECT (PRINT (CAR X)) (CDR X)`, in which case a `PROGN` is supplied.

Each operand can be the name of a function, in which case it is applied to the (last) i.v., e.g., `(for X in Y do PRINT when NUMBERP)` is the same as `(for X in Y do (PRINT X) when (NUMBERP X))`. Note that the i.v. need not be explicitly specified, e.g., `(in Y do PRINT when NUMBERP)` will work.

For i.s.types, e.g., `DO`, `COLLECT`, `JOIN`, the function is always applied to the first i.v. in the i.s., whether explicity named or not. For example, `(in Y as I from 1 to 10 do PRINT)` prints elements on `Y`, not integers between 1 and 10.

Note that this feature does not make much sense for `FOR`, `OLD`, `BIND`, `IN`, or `ON`, since they "operate" before the loop starts, when the i.v. may not even be bound.

In the case of `BY` in conjunction with `IN`, the function is applied to the current *tail* e.g., `(for X in Y by CDDR ...)` is the same as `(for X in Y by (CDDR X) ...)`.

While the exact translation of a loop depends on which operators are present, a `PROG` will always be used whenever the loop specifies dummy variables—if `BIND` appears, or there is more than one variable specified by a `FOR`, or a `GO`, `RETURN`, or a reference to the variable `$$VAL` appears in any of the operands. When `PROG` is used, the form of the translation is:

```
(PROG VARIABLES
    {initialize}
```

```
$$LP {eachtime}
    {test}
    {body}
$$ITERATE
    {aftertest}
    {update}
    (GO $$LP)
$$OUT {finalize}
    (RETURN $$VAL))
```

where {test} corresponds to that part of the loop that tests for termination and also for those iterations for which {body} is not going to be executed, (as indicated by a WHEN or UNLESS); {body} corresponds to the operand of the i.s.type, e.g., DO, COLLECT, etc.; {aftertest} corresponds to those tests for termination specified by REPEATWHILE or REPEATUNTIL; and {update} corresponds to that part that resets the tail, increments the counter, etc. in preparation for the next iteration. {initialize}, {finalize}, and {eachtime} correspond to the operands of FIRST, FINALLY, and EACHTIME, if any.

Since {body} always appears at the top level of the PROG, you can insert labels in {body}, and GO to them from within {body} or from other i.s. operands, e.g., (for X in Y first (GO A) do (FOO) A (FIE)). However, since {body} is dwimified as a list of forms, the label(s) should be added to the dummy variables for the iterative statement in order to prevent their being dwimified and possibly "corrected", e.g., (for X in Y bind A first (GO A) do (FOO) A (FIE)). You can also GO to $$LP, $$ITERATE, or $$OUT, or explicitly set $$VAL.

Errors in Iterative Statements

An error will be generated and an appropriate diagnostic printed if any of the following conditions hold:

1. Operator with null operand, i.e., two adjacent operators, as in (for X in Y until do ...)

2. Operand consisting of more than one form (except as operand to FIRST, FINALLY, or one of the i.s.types), e.g., (for X in Y (PRINT X) collect ...).

3. IN, ON, FROM, TO, or BY appear twice in same i.s.

4. Both IN and ON used on same i.v.

5. FROM or TO used with IN or ON on same i.v.

6. More than one i.s.type, e.g., a DO and a SUM.

In 3, 4, or 5, an error is not generated if an intervening AS occurs.

If an error occurs, the i.s. is left unchanged.

If no DO, COLLECT, JOIN or any of the other i.s.types are specified, CLISP will first attempt to find an operand consisting of more than one form, e.g., (for X in Y (PRINT X) when ATOM X ...), and in this case will insert a DO after the first form. (In this case, condition 2 is not considered to be met, and an error is not generated.) If CLISP cannot find such an operand, and no WHILE or UNTIL appears in the i.s., a warning message is printed: NO DO, COLLECT, OR JOIN: followed by the i.s.

Similarly, if no terminating condition is detected, i.e., no IN, ON, WHILE, UNTIL, TO, or a RETURN or GO, a warning message is printed: `Possible non-terminating iterative statement:` followed by the iterative statement. However, since the user may be planning to terminate the i.s. via an error, Control-E, or a RETFROM from a lower function, the i.s. is still translated.

Note: The error message is not printed if the value of CLISPI.S.GAG is T (initially NIL).

Defining New Iterative Statement Operators

The following function is available for defining new or redefining existing iterative statement operators:

(**I.S.OPR** *NAME FORM OTHERS EVALFLG*)   [Function]

> *NAME* is the name of the new i.s.opr. If *FORM* is a list, *NAME* will be a new *i.s.type*, and *FORM* its body.

> *OTHERS* is an (optional) list of additional i.s. operators and operands which will be added to the i.s. at the place where *NAME* appears. If *FORM* is NIL, *NAME* is a new i.s.opr defined entirely by *OTHERS*.

> In both *FORM* and *OTHERS*, the atom $$VAL can be used to reference the value to be returned by the i.s., I.V. to reference the current i.v., and BODY to reference *NAME*'s operand. In other words, the current i.v. will be substituted for all instances of I.V. and *NAME*'s operand will be substituted for all instances of BODY throughout *FORM* and *OTHERS*.

> If *EVALFLG* is T, *FORM* and *OTHERS* are evaluated at translation time, and their values used as described above. A dummy variable for use in translation that does not clash with a dummy variable already used by some other i.s. operators can be obtained by calling (GETDUMMYVAR). (GETDUMMYVAR T) will return a dummy variable and also insure that it is bound as a PROG variable in the translation.

> If *NAME* was previously an i.s.opr and is being redefined, the message (*NAME* REDEFINED) will be printed (unless DFNFLG=T), and all expressions using the i.s.opr *NAME* that have been translated will have their translations discarded.

> The following are some examples of how I.S.OPR could be called to define some existing i.s.oprs, and create some new ones:

```
COLLECT   (I.S.OPR 'COLLECT
             '(SETQ $$VAL (NCONC1 $$VAL BODY)))

    SUM   (I.S.OPR 'SUM
             '(SETQ $$VAL_(PLUS $$VAL BODY)
                '(FIRST (SETQ $$VAL0))

  NEVER   (I.S.OPR 'NEVER
             '(if BODY then
                (SETQ $$VAL NIL) (GO $$OUT))
```

> > Note: (if BODY then (RETURN NIL)) would exit from the i.s. immediately and therefore not execute the operations specified via a FINALLY (if any).

```
THEREIS   (I.S.OPR 'THEREIS
             '(if BODY then
                 (SETQ $$VAL I.V.) (GO $$OUT)))
```

RCOLLECT   To define RCOLLECT, a version of COLLECT which uses CONS instead of NCONC1 and then reverses the list of values:

```
(I.S.OPR 'RCOLLECT
  '(FINALLY (RETURN
      (DREVERSE $$VAL)))]
```

TCOLLECT   To define TCOLLECT, a version of COLLECT which uses TCONC:

```
(I.S.OPR 'TCOLLECT
  '(TCONC $$VAL BODY)
     '(FIRST (SETQ $$VAL (CONS))
         FINALLY (RETURN
             (CAR $$VAL)))]
```

```
PRODUCT   (I.S.OPR 'PRODUCT
             '(SETQ $$VAL $$VAL*BODY)
                 '(FIRST ($$VAL 1)))]
```

UPTO   To define UPTO, a version of TO whose operand is evaluated only once:

```
(I.S.OPR 'UPTO
  NIL
  '(BIND $$FOO←BODY TO $$FOO)]
```

TO   To redefine TO so that instead of recomputing *FORM* each iteration, a variable is bound to the value of *FORM*, and then that variable is used:

```
(I.S.OPR 'TO
  NIL
  '(BIND $$END FIRST
     (SETQ $$END BODY)
         ORIGINALTO $$END)]
```

Note the use of ORIGINAL to redefine TO in terms of its original definition. ORIGINAL is intended for use in redefining built-in operators, since their definitions are not accessible, and hence not directly modifiable. Thus if the operator had been defined by the user via I.S.OPR, ORIGINAL would not obtain its original definition. In this case, one presumably would simply modify the i.s.opr definition.

I.S.OPR can also be used to define synonyms for already defined i.s. operators by calling I.S.OPR with *FORM* an atom, e.g., (I.S.OPR 'WHERE 'WHEN) makes WHERE be the same as WHEN. Similarly, following (I.S.OPR 'ISTHERE 'THEREIS), one can write (ISTHERE ATOM IN Y), and following (I.S.OPR 'FIND 'FOR) and (I.S.OPR 'SUCHTHAT 'THEREIS), one can write (find X in Y suchthat X member Z). In the current system, WHERE is synonymous with WHEN, SUCHTHAT and ISTHERE with THEREIS, FIND with FOR, and THRU with TO.

If *FORM* is the atom `MODIFIER`, then *NAME* is defined as an i.s.opr which can immediately follow another i.s. operator (i.e., an error will not be generated, as described previously). *NAME* will not terminate the scope of the previous operator, and will be stripped off when `DWIMIFY` is called on its operand. `OLD` is an example of a `MODIFIER` type of operator. The `MODIFIER` feature allows the user to define i.s. operators similar to `OLD`, for use in conjunction with some other user defined i.s.opr which will produce the appropriate translation.

The file package command `I.S.OPRS` (Chapter 17) will dump the definition of i.s.oprs. `(I.S.OPRS PRODUCT  UPTO)` as a file package command will print suitable expressions so that these iterative statement operators will be (re)defined when the file is loaded.

[This page intentionally left blank]

Medley is designed to help you define and debug functions. Developing an applications program with Medley involves defining a number of functions in terms of the system primitives and other user-defined functions. Once defined, your functions may be used exactly like Interlisp primitive functions, so the programming process can be viewed as extending the Interlisp language to include the required functionality.

A function's definition specifies if the function has a fixed or variable number of arguments, whether these arguments are evaluated or not, the function argument names, and a series of forms which define the behavior of the function. For example:

```
(LAMBDA (X Y) (PRINT X) (PRINT Y))
```

This function has two evaluated arguments, x and y, and it will execute (PRINT X) and (PRINT Y) when evaluated. Other types of function definitions are described below.

A function is defined by putting an expr definition in the function definition cell of a symbol. There are a number of functions for accessing and setting function definition cells, but one usually defines a function with DEFINEQ (see the Defining Functions section below). For example:

```
← (DEFINEQ (FOO (LAMBDA (X Y) (PRINT X) (PRINT Y))))(FOO)
```

The expression above will define the function FOO to have the expr definition (LAMBDA (X Y) (PRINT X) (PRINT Y)). After being defined, this function may be evaluated just like any system function:

```
← (FOO 3 (IPLUS 3 4))
  3
  7
  7
```

Not all function definition cells contain expr definitions. The compiler (see the first page of Chapter 18) translates expr definitions into compiled code objects, which execute much faster. Interlisp provides a number of "function type functions" which determine how a given function is defined, the number and names of function arguments, etc. See the Function Type Functions section below.

Usually, functions are evaluated automatically when they appear within another function or when typed into Interlisp. However, sometimes it is useful to envoke the Interlisp interpreter explicitly to apply a given "functional argument" to some data. There are a number of functions which will apply a given function repeatedly. For example, MAPCAR will apply a function (or an expr definition) to all of the elements of a list, and return the values returned by the function:

```
← (MAPCAR '(1 2 3 4 5) '(LAMBDA (X) (ITIMES X X)))
  (1 4 9 16 25)
```

When using functional arguments, there are a number of problems which can arise, related to accessing free variables from within a function argument. Many times these problems can be solved using the function FUNCTION to create a FUNARG object.

The macro facility provides another way of specifying the behavior of a function (see the Macros section below). Macros are very useful when developing code which should run very quickly, which should be compiled differently than when it is interpreted, or which should run differently in different implementations of Interlisp.

Function Types

Interlisp functions are defined using list expressions called "expr definitions." An expr definition is a list of the form $(LAMBDA-WORD\ ARG-LIST\ FORM_1\ ...\ FORM_N)$. $LAMBDA-WORD$ determines whether the arguments to this function will be evaluated or not. $ARG-LIST$ determines the number and names of arguments. $FORM_1\ ...\ FORM_N$ are a series of forms to be evaluated after the arguments are bound to the local variables in $ARG-LIST$.

If $LAMBDA-WORD$ is the symbol LAMBDA, then the arguments to the function are evaluated. If $LAMBDA-WORD$ is the symbol NLAMBDA, then the arguments to the function are not evaluated. Functions which evaluate or don't evaluate their arguments are therefore known as "lambda" or "nlambda" functions, respectively.

If $ARG-LIST$ is NIL or a list of symbols, this indicates a function with a fixed number of arguments. Each symbol is the name of an argument for the function defined by this expression. The process of binding these symbols to the individual arguments is called "spreading" the arguments, and the function is called a "spread" function. If the argument list is any symbol other than NIL, this indicates a function with a variable number of arguments, known as a "nospread" function.

If $ARG-LIST$ is anything other than a symbol or a list of symbols, such as (LAMBDA "FOO" ...), attempting to use this expr definition will generate an Arg not symbol error. In addition, if NIL or T is used as an argument name, the error Attempt to bind NIL or T is generated.

These two parameters (lambda/nlambda and spread/nospread) may be specified independently, so there are four nain function types, known as lambda-spread, nlanbda-spread, lanbda-nospread, and nlambda-nospread functions. Each one has a different form and is used for a different purpose. These four function types are described more fully below.

For lambda-spread, lanbda-nospread, or nlambda-spread functions, there is an upper limit to the number of arguments that a function can have, based on the number of arguments that can be stored on the stack on any one function call. Currently, the limit is 80 arguments. If a function is called with more than that many arguments, the error Too many arguments occurs. However, nlambda-nospread functions can be called with an arbitrary number of arguments, since the arguments are not individually saved on the stack.

Lambda-Spread Functions

Lambda-spread functions take a fixed number of evaluated arguments. This is the most common function type. A lambda-spread expr definition has the form:

$$(LAMBDA\ (ARG_1\ ...\ ARG_M)\ FORM_1\ ...\ FORM_N)$$

The argument list $(ARG_1\ ...\ ARG_M)$ is a list of symbols that gives the number and names of the formal arguments to the function. If the argument list is ( ) or NIL, this indicates that the function takes no arguments. When a lambda-spread function is applied to some arguments, the arguments are evaluated, and bound to the local variables $ARG_1\ ...\ ARG_M$. Then, $FORM_1\ ...\ FORM_N$ are evaluated in order, and the value of the function is the value of $FORM_N$.

```
← (DEFINEQ (FOO (LAMBDA (X Y) (LIST X Y))))
    (FOO)
← (FOO 99 (PLUS 3 4))
    (99 7)
```

In the above example, the function FOO defined by (LAMBDA (X Y) (LIST X Y)) is applied to the arguments 99 and (PLUS 3 4). These arguments are evaluated (giving 99 and 7), the local variable X is bound to 99 and Y to 7, (LIST X Y) is evaluated, returning (99 7), and this is returned as the value of the function.

A standard feature of the Interlisp system is that no error occurs if a spread function is called with too many or too few arguments. If a function is called with too many argumnents, the extra arguments are evaluated but ignored. If a function is called with too few arguments, the unsupplied ones will be delivered as NIL. In fact, a spread function cannot distinguish between being given NIL as an argument, and not being given that argument, e.g., (FOO) and (FOO NIL) are exactly the same for spread functions. If it is necessary to distinguish between these two cases, use an nlambda function and explicitly evaluate the arguments with the EVAL function.

Nlambda-Spread Functions

Nlambda-spread functions take a fixed number of unevaluated arguments. An nlambda-spread expr definition has the form:

$$(\text{NLAMBDA } (ARG_1 \ ... \ ARG_M) \ FORM_1 \ ... \ FORM_N)$$

Nlambda-spread functions are evaluated similarly to lanbda-spread functions, except that the arguments are not evaluated before being bound to the variables $ARG_1 \ ... \ ARG_M$.

```
← (DEFINEQ (FOO (NLAMBDA (X Y) (LIST X Y))))
  (FOO)
← (FOO 99 (PLUS 3 4))
  (99 (PLUS 3 4))
```

In the above example, the function FOO defined by (NLAMBDA (X Y) (LIST X Y)) is applied to the arguments 99 and (PLUS 3 4). These arguments are unevaluated to X and Y. (LIST X Y) is evaluated, returning (99 (PLUS 3 4)), and this is returned as the value of the function.

Functions can be defined so that all of their arguments are evaluated (lambda functions) or none are evaluated (nlambda functions). If it is desirable to write a function which only evaluates some of its arguments (e.g., SETQ), the functions should be defined as an nlambda, with some arguments explicitly evaluated using the function EVAL. If this is done, the user should put the symbol EVAL on the property list of the function under the property INFO. This informs various system packages, such as DWIM, CLISP, and Masterscope, that this function in fact does evaluate its arguments, even though it is an nlambda.

Warning: A frequent problem that occurs when evaluating arguments to nlambda functions with EVAL is that the form being evaluated may reference variables that are not accessible within the nlambda function. This is usually not a problem when interpreting code, but when the code is compiled, the values of "local" variables may not be accessible on the stack (see Chapter 18). The system nlambda functions that evaluate their arguments (such as SETQ) are expanded in-line by the compiler, so this is not a problem. Using the macro facility is recommended in cases where it is necessary to evaluate some arguments to an nlambda function.

Lambda-Nospread Functions

Lambda-nospread functions take a variable number of evaluated arguments. A lambda-nospread expr definition has the form:

$$(\text{LAMBDA } VAR \ FORM_1 \ ... \ FORM_N)$$

VAR may be any symbol, except NIL and T. When a lambda-nospread function is applied to some arguments, each of these arguments is evaluated and the values stored on the stack. *VAR* is then bound to the number of arguments which have been evaluated. For example, if FOO is defined by (LAMBDA X ...), when (FOO A B C) is evaluated, A, B, and C are evaluated and X is bound to 3. *VAR* should never be reset

The following functions are used for accessing the arguments of lambda-nospread functions.

(**ARG** *VAR M*)   [NLambda Function]

> Returns the *M*th argument for the lambda-nospread function whose argument list is *VAR*. *VAR* is the name of the atomic argument list to a lambda-nospread function, and is not evaluated. *M* is the number of the desired argument, and is evaluated. The value of ARG is undefined for *M* less than or equal to 0 or greater than the value of *VAR*.

(**SETARG** *VAR M X*)   [NLambda Function]

> Sets the *M*th argument for the lambda-nospread function whose argument list is *VAR* to *X*. *VAR* is not evaluated; *M* and *X* are evaluated. *M* should be between 1 and the value of *VAR*.

In the example below, the function FOO is defined to collect and return a list of all of the evaluated arguments it is given (the value of the for statement).

```
← (DEFINEQ (FOO
  (LAMBDA X (for ARGNUM from 1 to X collect (ARG X ARGNUM)]
   (FOO)
← (FOO 99 (PLUS 3 4))
   (99 7)
← (FOO 99 (PLUS 3 4)(TIMES 3 4)))
   (99 7 12)
```

NLambda-Nospread Functions

Nlambda-nospread functions take a variable number of unevaluated arguments. An nlambda-nospread expr definition has the form:

(NLAMBDA *VAR FORM$_1$ ... FORM$_N$*)

VAR may be any symbol, except NIL and T. Though similar in form to lambda-nospread expr definitions, an nlambda-nospread is evaluated quite differently. When an nlambda-nospread function is applied to some arguments, *VAR* is simply bound to a list of the unevaluated arguments. The user may pick apart this list, and evaluate different arguments.

In the example below, FOO is defined to return the reverse of the list of arguments it is given (unevaluated):

```
← (DEFINEQ (FOO (NLAMBDA X (REVERSE X))))
   (FOO)
← (FOO 99 (PLUS 3 4))
   ((PLUS 3 4) 99)
← (FOO 99 (PLUS 3 4)(TIMES 3 4))
   (TIMES 3 4)(PLUS 3 4) 99)
```

The warning about evaluating arguments to nlambda functions also applies to nlambda-nospread function.

Compiled Functions

Functions defined by expr definitions can be compiled by the Interlisp compiler (see Chapter 18). The compiler produces compiled code objects (of data type CCODEP) which execute more quickly than the corresponding expr definition code. Functions defined by compiled code objects may have the same four types as expr definitions (lambda/nlambda, spread/nospread). Functions created by the compiler are referred to as compiled functions.

Function Type Functions

There are a variety of functions used for examining the type, argument list, etc. of functions. These functions may be given either a symbol (in which case they obtain the function definition from the definition cell), or a function definition itself.

(**FNTYP** *FN*) [Function]

> Returns NIL if *FN* is not a function definition or the name of a defined function. Otherwise, FNTYP returns one of the following symbols, depending on the type of function definition.
>
>| | |
>|---|---|
>| EXPR | Lambda-spread expr definition |
>| CEXPR | Lambda-spread compiled definition |
>| FEXPR | Nlambda-spread expr definition |
>| CFEXPR | Nlambda-spread compiled definition |
>| EXPR* | Lambda-nospread expr definition |
>| CEXPR* | Lambda-nospread compiled definition |
>| FEXPR* | Nlambda-nospread expr definition |
>| CFEXPR* | Nlambda-nospread compiled definition |
>| FUNARG | FNTYP returns the symbol FUNARG if *FN* is a FUNARG expression. |
>
> EXP, FEXPR, EXPR*, and FEXPR* indicate that *FN* is defined by an expr definition. CEXPR, CFEXPR, CEXPR*, and CFEXPR* indicate that *FN* is defined by a compiled definition, as indicated by the prefix C. The suffix * indicates that *FN* has an indefinite number of arguments, i.e., is a nospread function. The prefix F indicates unevaluated arguments. Thus, for example, a CFEXPR* is a compiled nospread nlambda function.

(**EXPRP** *FN*) [Function]

> Returns T if (FNTYP *FN*) is EXPR, FEXPR, EXPR*, or FEXPR*; NIL otherwise. However, (EXPRP *FN*) is also true if *FN* is (has) a list definition, even if it does not begin with LAMBDA or NLAMBDA. In other words, EXPRP is not quite as selective as FNTYP.

(**CCODEP** *FN*) [Function]

> Returns T if (FNTYP *FN*) is either CEXPR, CFEXPR, CEXPR*, or CFEXPR*; NIL otherwise.

(**ARGTYPE** *FN*) [Function]

> *FN* is the name of a function or its definition. ARGTYPE returns 0, 1, 2, or 3, or NIL if *FN* is not a function. ARGTYPE corresponds to the rows of FNTYPs. The interpretation of this value is as follows:
>
>| | |
>|---|---|
>| 0 | Lambda-spread function (EXPR, CEXPR) |
>| 1 | Nlambda-spread function (FEXPR, CFEXPR) |

2   Lambda-nospread function (EXPR*, CEXPR*)

3   Nlambda-nospread function (FEXPR*, CFEXPR*)

(**NARGS** *FN*)   [Function]

Returns the number of arguments of *FN*, or NIL if *FN* is not a function. If *FN* is a nospread function, the value of NARGS is 1.

(**ARGLIST** *FN*)   [Function]

Returns the "argument list" for *FN*. Note that the "argument list" is a symbol for nospread functions. Since NIL is a possible value for ARGLIST, the error Args not available is generated if *FN* is not a function.

If *FN* is a compiled function, the argument list is constructed, i.e., each call to ARGLIST requires making a new list. For functions defined by expr definitions, lists beginning with LAMBDA or NLAMBDA, the argument list is simply CADR of GETD. If *FN* has an expr definition, and CAR of the definition is not LAMBDA or NLAMBDA, ARGLIST will check to see if CAR of the definition is a member of LAMBDASPLST (see Chapter 20). If it is, ARGLIST presumes this is a function object the user is defining via DWIMUSERFORMS, and simply returns CADR of the definition as its argument list. Otherwise ARGLIST generates an error as described above.

(**SMARTARGLIST** *FN EXPLAINFLG TAIL*)   [Function]

A "smart" version of ARGLIST that tries various strategies to get the arglist of *FN*.

First SMARTARGLIST checks the property list of *FN* under the property ARGNAMES. For spread functions, the argument list itself is stored. For nospread functions, the form is (NIL ARGLIST$_1$. ARGLIST$_2$), where ARGLIST$_1$ is the value SMARTARGLIST should return when *EXPLAINFLG* = T, and ARGLIST$_2$ the value when *EXPLAINFLG* = NIL. For example, (GETPROP 'DEFINEQ 'ARGNAMES) = (NIL (X1 X1 ... XN) . X). This allows the user to specify special argument lists.

Second, if *FN* is not defined as a function, SMARTARGLIST attempts spelling correction on *FN* by calling FNCHECK (see Chapter 20), passing *TAIL* to be used for the call to FIXSPELL. If unsuccessful, the FN Not a function error will be generated.

Third, if *FN* is known to the file package (see Chapter 17) but not loaded in, SMARTARGLIST will obtain the arglist information from the file.

Otherwise, SMARTARGLIST simply returns (ARGLIST *FN*).

SMARTARGLIST is used by BREAK (see Chapter 15) and ADVISE with *EXPLAINFLG* = NIL for constructing equivalent expr definitions, and by the TTYIN in-line command ?= (see Chapter 26), with *EXPLAINFLG* = T.

Defining Functions

Function definitions are stored in a "function definition cell" associated with each symbol. This cell is directly accessible via the two functions PUTD and GETD (see below), but it is usually easier to define functions with DEFINEQ:

(**DEFINEQ** $X_1 X_2$ . . . X_N)   [NLambda NoSpread Function]

> DEFINEQ is the function normally used for defining functions. It takes an indefinite number of arguments which are not evaluated. Each X_i must be a list defining one function, of the form (NAME DEFINITION). For example:
>
>   (DEFINEQ (DOUBLE (LAMBDA (X) (IPLUS X X))))
>
> The above expression will define the function DOUBLE with the expr definition (LAMBDA (X) (IPLUS X X)). X_i may also have the form (NAME ARGS . DEF-BODY), in which case an appropriate lambda expr definition will be constructed. Therefore, the above expression is exactly the same as:
>
>   (DEFINEQ (DOUBLE (X) (IPLUS X X)))
>
> Note that this alternate form can only be used for lambda functions. The first form must be used to define an nlambda function.
>
> DEFINEQ returns a list of the names of the functions defined.

(**DEFINE** X —)   [Function]

> Lambda-spread version of DEFINEQ. Each element of the list X is itself a list either of the form (NAME DEFINITION) or (NAME ARGS . DEF-BODY). DEFINE will generate an error, Incorrect defining form on encountering an atom where a defining list is expected.

DEFINE and DEFINEQ operate correctly if the function is already defined and BROKEN, ADVISED, or BROKEN-IN.

For expressions involving type-in only, if the time stamp facility is enabled (see the Time Stamps section of Chapter 16), both DEFINE and DEFINEQ stamp the definition with your initials and date.

UNSAFE.TO.MODIFY.FNS   [Variable]

> Value is a list of functions that should not be redefined, because doing so may cause unusual bugs (or crash the system!). If you try to modify a function on this list (using DEFINEQ, TRACE, etc), the system prints Warning: XXX may be unsafe to modify -- continue? If you type Yes, the function is modified, otherwise an error occurs. This provides a measure of safety for novices who may accidently redefine important system functions. You can add your own functions onto this list.
>
> By convention, all functions starting with the character backslash ("\") are system internal functions, which you should never redefine or modify. Backslash functions are not on UNSAFE.TO.MODIFY.FNS, so trying to redefine them will not cause a warning.

DFNFLG   [Variable]

> DFNFLG is a global variable that affects the operation of DEFINEQ and DEFINE. If DFNFLG=NIL, an attempt to *redefine* a function FN will cause DEFINE to print the message (FN REDEFINED) and to save the old definition of FN using SAVEDEF (see the Functions for Manipulating Typed Definitions section of Chapter 17) before redefining it (except if the old and new definitions are EQUAL, in which case the effect is simply a no-op). If DFNFLG=T, the function is simply redefined. If DFNFLG=PROP or ALLPROP, the new definition is stored on the property list under the property EXPR. ALLPROP also affects the operation of RPAQQ and RPAQ (see the Functions Used Within Source Files section of Chapter 17). DFNFLG is initially NIL.

DFNFLG is reset by LOAD (see the Loading Files section of Chapter 17) to enable various ways of handling the defining of functions and setting of variables when loading a file. For most applications, the user will not reset DFNFLG directly.

Note: The compiler does *not* respect the value of DFNFLG when it redefines functions to their compiled definitions (see the first page of Chapter 18). Therefore, if you set DFNFLG to PROP to completely avoid inadvertantly redefining something in your running system, you *must* use compile mode F, not ST.

Note that the functions SAVEDEF and UNSAVEDEF (see the Functions for Manipulating Typed Definitions section of Chapter 17) can be useful for "saving" and restoring function definitions from property lists.

(**GETD** *FN*)   [Function]

Returns the function definition of *FN*. Returns NIL if *FN* is not a symbol, or has no definition.

GETD of a compiled function constructs a pointer to the definition, with the result that two successive calls do not necessarily produce EQ results. EQP or EQUAL must be used to compare compiled definitions.

(**PUTD** *FN DEF –*)   [Function]

Puts *DEF* into *FN*'s function cell, and returns *DEF*. Generates an error, Arg not symbol, if *FN* is not a symbol. Generates an error, Illegal arg, if *DEF* is a string, number, or a symbol other than NIL.

(**MOVD** *FROM TO COPYFLG –*)   [Function]

Moves the definition of *FROM* to *TO*, i.e., redefines *TO*. If *COPYFLG* = T, a COPY of the definition of *FROM* is used. *COPYFLG* =T is only meaningful for expr definitions, although MOVD works for compiled functions as well. MOVD returns *TO*.

COPYDEF (see the Functions for Manipulating Typed Definitions section of Chapter 17) is a higher-level function that not only moves expr definitions, but works also for variables, records, etc.

(**MOVD?** *FROM TO COPYFLG –*)   [Function]

If *TO* is not defined, same as (MOVD *FROM TO COPYFLG*). Otherwise, does nothing and returns NIL.

Function Evaluation

Usually, function application is done automatically by the Interlisp interpreter. If a form is typed into Interlisp whose CAR is a function, this function is applied to the arguments in the CDR of the form. These arguments are evaluated or not, and bound to the funcion parameters, as determined by the type of the function, and the body of the function is evaluated. This sequence is repeated as each form in the body of the function is evaluated.

There are some situations where it is necessary to explicitly call the evaluator, and Interlisp supplies a number of functions that will do this. These functions take "functional arguments," which may either

be symbols with function definitions, or expr definition forms such as (LAMBDA (X)...), or FUNARG expressions.

(**APPLY** *FN ARGLIST* −)   [Function]

> Applies the function *FN* to the arguments in the list *ARGLIST*, and returns its value. APPLY is a lambda function, so its arguments are evaluated, but the individual elements of *ARGLIST* are not evaluated. Therefore, lambda and nlambda functions are treated the same by APPLY—lambda functions take their arguments from *ARGLIST* without evaluating them. For example:
>
> ```
> ←  (APPLY 'APPEND '((PLUS 1 2 3)(4 5 6)))
>       (PLUS 1 2 3 4 5 6)
> ```
>
> Note that *FN* may explicitly evaluate one or more of its arguments itself. For example, the system function SETQ is an nlambda function that explicitly evaluates its second argument. Therefore, (APPLY 'SETQ '(FOO (ADD1 3))) will set FOO to 4, instead of setting it to the expression (ADD1 3).
>
> APPLY can be used for manipulating expr definitions. For example:
>
> ```
> ←  (APPLY '(LAMBDA (X Y)(ITIMES X Y)) '(3 4)))
>       12
> ```

(**APPLY*** *FN ARG₁ ARG₂ . . . ARGₙ*)   [NoSpread Function]

> Nospread version of APPLY. Applies the function *FN* to the arguments *ARG₁ ARG₂* . . . *ARGₙ*. For example:
>
> ```
> ←  (APPLY 'APPEND '(PLUS 1 2 3)(4 5 6))
>       (PLUS 1 2 3 4 5 6)
> ```

(**EVAL** *X*−)   [Function]

> EVAL evaluates the expression *X* and returns this value, i.e., EVAL provides a way of calling the Interlisp interpreter. Note that EVAL is itself a lambda function, so its argument is first evaluated, e.g.:
>
> ```
> ←  (SETQ FOO 'ADD1 3)))
>       (ADD1 3)
> ←(EVAL FOO)
>       4
> ←(EVAL 'FOO)
>       (ADD1 3)
> ```

(**QUOTE** *X*)   [Nlambda NoSpread Function]

> QUOTE prevents its arguments from being evaluated. Its value is *X* itself, e.g., (QUOTE FOO) is FOO.
>
> Interlisp functions can either evaluate or not evaluate their arguments. QUOTE can be used in those cases where it is desirable to specify arguments unevaluated.
>
> The single-quote character (') is defined with a read macro so it returns the next **expression, wrapped in a call to** QUOTE **(see Chapter 25).** For example, 'FOO reads as (QUOTE FOO). This is the form used for examples in this manual.
>
> Since giving QUOTE more than one argument is almost always a parenthese error, and one that would otherewise go undetected, QUOTE itself generates an error in this case, Parenthesis error.

(**KWOTE** *X*)   [Function]

Value is an expression which, when evaluated, yields *X*. If *X* is NIL or a number, this is *X* itself. Otherwise (LIST (QUOTE QUOTE) X). For example:

```
(KWOTE 5) => 5
(KWOTE (CONS 'A 'B)) => (QUOTE (A.B))
```

(**NLAMBDA.ARGS** *X*)   [Function]

This function interprets its argument as a list of unevaluated nlambda arguments. If any of the elements in this list are of the form (QUOTE...), the enclosing QUOTE is stripped off. Actually, NLAMBDA.ARGS stops processing the list after the first non-quoted argument. Therefore, whereas (NLAMBDA.ARGS '((QUOTE FOO) BAR)) -> (FOO BAR), (NLAMBDA.ARGS '(FOO (QUOTE BAR))) -> (FOO (QUOTE BAR)).

NLAMBDA.ARGS is alled by a number of nlambda functions in the system, to interpret their arguments. For instance, the function BREAK calls NLAMBDA.ARGS so that (BREAK 'FOO) will break the function FOO, rather than the function QUOTE.

(**EVALA** *X* *A*)   [Function]

Simulates association list variable lookup. *X* is a form, *A* is a list of the form:

$$((NAME_1 . VAL_1) (NAME_2 . VAL_2)... (NAME_N . VAL_N))$$

The variable names and values in *A* are "spread" on the stack, and then *X* is evaluated. Therefore, any variables appearing free in *X* that also appears as CAR of an element of *A* will be given the value on the CDR of that element.

(**DEFEVAL** *TYPE FN*)   [Function]

Specifies how a datum of a particular type is to be evaluated. Intended primarily for user-defined data types, but works for all data types except lists, literal atoms, and numbers. *TYPE* is a type name. *FN* is a function object, i.e., name of a function or a lambda expression. Whenever the interpreter encounters a datum of the indicated type, *FN* is applied to the datum and its value returned as the result of the evaluation. DEFEVAL returns the previous evaling function for this type. If *FN* = NIL, DEFEVAL returns the current evaling function without changing it. If *FN* = T, the evaling functions is set back to the system default (which for all data types except lists is to return the datum itself).

COMPILETYPELST (see Chapter 18) permits the user to specify how a datum of a particular type is to be compiled.

(**EVALHOOK** *FORM EVALHOOKFN*)   [Function]

EVALHOOK evaluates the expression *FORM*, and returns its value. While evaluating *FORM*, the function EVAL behaves in a special way. Whenever a list other than *FORM* itself is to be evaluated, whether implicitly or via an explicit call to EVAL, *EVALHOOKFN* is invoked (it should be a function), with the form to be evaluated as its argument. *EVALHOOKFN* is then responsible for evaluating the form. Whatever is returned is assume to be the result of evaluating the form. During the execution of *EVALHOOKFN*, this special evaluation is turned off. (Note that EVALHOOK does not affect the evaluations of variables, only of lists).

Here is an example of a simple tracing routine that uses the EVALHOOK feature:

```
←(DEFINEQ (PRINTHOOK (FORM)
     (printout T "eval: "FORM T)
     (EVALHOOK FORM (FUNCTION PRINTHOOK
     (PRINTHOOK)
```

Using PRINTHOOK, one might see the following interaction:

```
←(EVALHOOK '(LIST (CONS 1 2)(CONS 3 4)) 'PRINTHOOK)
     eval: (CONS 1 2)
     eval: (CONS 3 4)
     ((1.2)(3.4))
```

Iterating and Mapping Functions

The functions below are used to evaluate a form or apply a function repeatedly. RPT, RPTQ, and FRPTQ evaluate an expression a specified number of time. MAP, MAPCAR, MAPLIST, etc., apply a given function repeatedly to different elements of a list, possibly constructing another list.

These functions allow efficient iterative computations, but they are difficult to use. For programming iterative computations, it is usually better to use the CLISP Iterative Statement facility (see Chapter 9), which provides a more general and complete facility for expressing iterative statements. Whenever possible, CLISP transltes iterative statements into expressions using the functions below, so there is no efficiency loss.

(**RPT** N $FORM$)   [Function]

Evaluates the expression $FORM$, N times. Returns the value of the last evaluation. If N is less than or equal to 0, $FORM$ is not evaluated, and RPT returns NIL.

Before each evaluation, the local variable RPTN is bound to the number of evaluations yet to take place. This variable can be referenced within $FORM$. For example, (RPT 10 '(PRINT RPTN)) will print the numbers 10, 9...1, and return 1.

(**RPTQ** N $FORM_1$ $FORM_2$... $FORM_N$)   [NLambda NoSpread Function]

Nlambda-nospread version of RPT: N is evaluated, $FORM_i$ are not. Returns the value of the last evaluation of $FORM_N$.

(**FRPTQ** N $FORM_1$ $FORM_2$... $FORM_N$)   [NLambda NoSpread Function]

Faster version of RPTQ. Does not bind RPTN.

(**MAP** MAP_X $MAPFN_1$ $MAPFN_2$)   [Function]

If $MAPFN_2$ is NIL, MAP applies the function $MAPFN_1$ to successive tails of the list MAP_X. That is, first it computes ($MAPFN_1$ MAP_X), and then ($MAPFN_1$ (CDR MAP_X)), etc., until MAP_X becomes a non-list. If $MAPFN_2$ is provided, ($MAPFN_2$ MAP_X) is used instead of (CDR MAP_X) for the next call for $MAPFN_1$, e.g., if $MAPFN_2$ were CDDR, alternate elements of the list would be skipped. MAP returns NIL.

(**MAPC** MAP_X $MAPFN_1$ $MAPFN_2$)   [Function]

Identical to MAP, except that ($MAPFN_1$ (CAR MAP_X)) is computed at each iteration instead of ($MAPFN_1$ MAP_X), i.e., MAPC works on elements, MAP on tails. MAPC returns NIL.

(**MAPLIST** MAP_X $MAPFN_1$ $MAPFN_2$) [Function]

Successively computes the same values that MAP would compute, and returns a list consisting of those values.

(**MAPCAR** MAP_X $MAPFN_1$ $MAPFN_2$) [Function]

Computes the same values that MAPC would compute, and returns a list consisting of those values, e.g., (MAPCAR X 'FNTYP) is a list of FNTYPs for each element on X.

(**MAPCON** MAP_X $MAPFN_1$ $MAPFN_2$) [Function]

Computes the same values that MAP and MAPLIST but NCONCs these values to form a list which it returns.

(**MAPCONC** MAP_X $MAPFN_1$ $MAPFN_2$) [Function]

Computes the same values that MAPC and MAPCAR, but NCONCs the values to form a list which it returns.

Note that MAPCAR creates a new list which is a mapping of the old list in that each element of the new list is the result of applying a function to the corresponding element on the original list. MAPCONC is used when there are a variable number of elements (including none) to be inserted at each iteration. Examples:

```
(MAPCONC '(A B C NIL D NIL) '(LAMBDA (Y)(if (NULL Y) then NIL
   else (LIST Y)))) = > (A B C D)
```

This MAPCONC returns a list consisting of MAP_X with all NILs removed.

```
(MAPCONC '((A B) C (D E F)(G) H I) '(LAMBDA (Y)(if (LISP Y) then Y
   else NIL))) = > (A B D E F G)
```

This MAPCONC returns a linear list consisting of all the lists on MAP_X.

Since MAPCONC uses NCONC to string the corresponding lists together, in this example the original list will be altered to be ((A B C D E F G) C (D E F G)(G) H I). If this is an undesirable side effect, the functional argument to MAPCONC should return instead a top level copy of the lists, i.e., (LAMBDA (Y) (if (LISTP Y) then (APPERND Y) else NIL))).

(**MAP2C** MAP_X MAP_Y $MAPFN_1$ $MAPFN_2$) [Function]

Identical to MAPC except $MAPFN_1$ is a function of two arguments, and ($MAPFN_1$ (CAR MAP_X)(CAR MAP_Y)) is computed at each iteration. Terminates when either MAP_X or MAP_Y is a non-list.

$MAPFN_2$ is still a function of one argument, and is applied twice on each iteration; ($MAPFN_2$ MAP_X) gives the new MAP_X, ($MAPFN_2$ MAP_Y) the new MAP_Y. CDR is used if $MAPFN_2$ is not supplied, i.e., is NIL.

(**MAP2CAR** MAP_X MAP_Y $MAPFN_1$ $MAPFN_2$) [Function]

Identical to MAPCAR except $MAPFN_1$ is a function of two arguments, and ($MAPFN_1$ (CAR MAP_X)(CAR MAP_Y)) is used to assemble the new list. Terminates when either MAP_X or MAP_Y is a non-list.

($\textbf{SUBSET}$ MAP_X $MAPFN_1$ $MAPFN_2$) [Function]

Applies $MAPFN_1$ to elements of MAP_X and returns a list of those elements for which this application is non-NIL, e.g.:

(SUBSET '(A B 3 C 4) 'NUMBERP) = (3 4)

$MAPFN_2$ plays the same role as with MAP, MAPC, et al.

($\textbf{EVERY}$ $EVERY_X$ $EVERYFN_1$ $EVERYFN_2$) [Function]

Returns T if the result of applying $EVERYFN_1$ to each element in $EVERY_X$ is true, otherwise NIL. For example, (EVERY '(X Y Z) 'ATOM) => T.

EVERY operates by evaluating ($EVERYFN_1$ (CAR $EVERY_X$) $EVERY_X$). The second argument is passed to $EVERYFN_1$ so that it can look at the next element on $EVERY_X$ if necessary. If $EVERYFN_1$ yields NIL, EVERY immediately returns NIL. Otherwise, EVERY computes ($EVERYFN_2$ $EVERY_X$), or (CDR $EVERY_X$) if $EVERYFN_2$ = NIL, and uses this as the "new" $EVERY_X$, and the process continues. For example (EVERY X 'ATOM 'CDDR) is true if every other element of X is atomic.

($\textbf{SOME}$ $SOME_X$ $SOMEFN_1$ $SOMEFN_2$) [Function]

Returns the tail of $SOME_X$ beginning with the first element that satisfies $SOMEFN_1$, i.e., for which $SOMEFN_1$ applied to that element is true. Value is NIL if no such element exists. (SOME X '(LAMBDA (Z) (EQUAL Z Y))) is equivalent to (MEMBER Y X). SOME operates analogously to EVERY. At each stage, ($SOMEFN_1$ (CAR $SOME_X$) $SOME_X$) is computed, and if this not NIL, $SOME_X$ is returned as the value of SOME. Otherwise, ($SOMEFN_2$ $SOME_X$) is computed, or (CDR $SOME_X$) if $SOMEFN_2$ = NIL, and used for the next $SOME_X$.

($\textbf{NOTANY}$ $SOME_X$ $SOMEFN_1$ $SOMEFN_2$) [Function]

(NOT (SOME $SOME_X$ $SOMEFN_1$ $SOMEFN_2$)).

($\textbf{NOTEVERY}$ $EVERY_X$ $EVERYFN_1$ $EVERYFN_2$) [Function]

(NOT (EVERY $EVERY_X$ $EVERYFN_1$ $EVERYFN_2$)).

($\textbf{MAPRINT}$ LST $FILE$ $LEFT$ $RIGHT$ SEP PFN $LISPXPRINTFLG$) [Function]

A general printing function. For each element of the list LST, applies PFN to the element, and $FILE$. If PFN is NIL, PRIN1 is used. Between each application MAPRINT performs PRIN1 of SEP (or " " if SEP = NIL). If $LEFT$ is given, it is printed (using PRIN1) initially; if $RIGHT$ is given, it is printed (using PRIN1) at the end.

For example, (MAPRINT X NIL '%('%)) is equivalent to PRIN1 for lists. To print a list with commas between each element and a final "." one could use (MAPRINT X T NIL '%. '%,).

If $LISPXPRINTFLG$ = T, LISPXPRIN1 (see Chapter 13) is used instead of PRIN1.

Functional Arguments

The functions that call the Interlisp-D evaluator take "functional arguments," which may be symbols with function definitions, or expr definition forms such as (LAMBDA (X) ...).

The following functions are useful when one wants to supply a functional argument which will always return NIL, T, or 0. Note that the arguments X_1 ... X_N to these functions are evaluated, though they are not used.

   (**NILL** X_1 ... X_N)   [NoSpread Function]

      Returns NIL.

   (**TRUE** X_1 ... X_N)   [NoSpread Function]

      Returns T.

   (**ZERO** X_1 ... X_N)   [NoSpread Function]

      Returns 0.

When using expr definitions as function arguments, they should be enclosed within the function FUNCTION rather than QUOTE, so that they will be compiled as separate functions.

   (**FUNCTION** *FN ENV*)   [NLambda Function]

    If *ENV* = NIL, FUNCTION is the same as QUOTE, except that it is treated differently when compiled. Consider the function definition:

    (DEFINEQ (FOO (LST)(FIE LST (FUNCTION (LAMBDA (Z)(ITIMES Z Z))]

    FOO calls the function FIE with the value of LST and the expr definition (LAMBDA (Z)(LIST (CAR Z))).

    If FOO is run interpreted, it does not make any difference whether FUNCTION or QUOTE is used. However, when FOO is compiled, if FUNCTION is used the compiler will define and compile the expr definition as an auxiliary function (see Chapter 18). The compiled expr definition will run considerably faster, which can make a big difference if it is applied repeatedly.

    Compiling FUNCTION will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (MAPCAR, MAPLIST, etc.).

    If *ENV* is not NIL, it can be a list of variables that are (presumably) used freely by *FN*. *ENV* can also be an atom, in which case it is evaluated, and the value interpreted as described above.

Macros

Macros provide an alternative way of specifying the action of a function. Whereas function definitions are evaluated with a "function call", which involves binding variables and other housekeeping tasks, macros are evaluated by *translating* one Interlisp form into another, which is then evaluated.

A symbol may have both a function definition and a macro definition. When a form is evaluated by the interpreter, if the CAR has a function definition, it is used (with a function call), otherwise if it has a

macro definition, then that is used. However, when a form is compiled, the CAR is checked for a macro definition first, and only if there isn't one is the function definition compiled. This allows functions that behave differently when compiled and interpreted. For example, it is possible to define a function that, when interpreted, has a function definition that is slow and has a lot of error checks, for use when debugging a system. This function could also have a macro definition that defines a fast version of the function, which is used when the debugged system is compiled.

Macro definitions are represented by lists that are stored on the property list of a symbol. Macros are often used for functions that should be compiled differently in different Interlisp implementations, and the exact property name a macro definition is stored under determines whether it should be used in a particular implementation. The global variable MACROPROPS contains a list of all possible macro property names which should be saved by the MACROS file package command. Typical macro property names are DMACRO for Interlisp-D, 10MACRO for Interlisp-10, VAXMACRO for Interlisp-VAX, JMACRO for Interlisp-Jerico, and MACRO for "implementation independent" macros. The global variable COMPILERMACROPROPS is a list of macro property names. Interlisp determines whether a symbol has a macro definition by checking these property names, in order, and using the first non-NIL property value as the macro definition. In Interlisp-D this list contains DMACRO and MACRO in that order so that DMACROs will override the implementation-independent MACRO properties. In general, use a DMACRO property for macros that are to be used only in Interlisp-D, use 10MACRO for macros that are to be used only in Interlisp-10, and use MACRO for macros that are to affect both systems.

Macro definitions can take the following forms:

(**LAMBDA ...**)
(**NLAMBDA ...**) A function can be made to compile open by giving it a macro definition of the form (LAMBDA ...) or (NLAMBDA ...), e.g., (LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X)))) for ABS. The effect is as if the macro definition were written in place of the function wherever it appears in a function being compiled, i.e., it compiles as a lambda or nlambda expression. This saves the time necessary to call the function at the price of more compiled code generated in-line.

(**NIL** *EXPRESSION*)
(**LIST** *EXPRESSION*) "Substitution" macro. Each argument in the form being evaluated or compiled is substituted for the corresponding atom in LIST, and the result of the substitution is used instead of the form. For example, if the macro definition of ADD1 is ((X) (IPLUS X 1)), then, (ADD1 (CAR Y)) is compiled as (IPLUS (CAR Y) 1).

Note that ABS could be defined by the substitution macro ((X) (COND ((GREATERP X 0) X) (T (MINUS X)))). In this case, however, (ABS (FOO X)) would compile as

```
(COND ((GREATERP (FOO X) 0)
        (FOO X))
 (T (MINUS (FOO X))))
```

and (FOO X) would be evaluated two times. (Code to evaluate (FOO X) would be generated three times.)

(**OPENLAMBDA** *ARGS BODY*) This is a cross between substitution and LAMBDA macros. When the compiler processes an OPENLAMBDA, it attempts to substitute the actual arguments for the formals wherever this preserves the frequency and

order of evaluation that would have resulted from a LAMBDA expression, and produces a LAMBDA binding only for those that require it.

Note: OPENLAMBDA assumes that it can substitute literally the actual arguments for the formal arguments in the body of the macro if the actual is side-effect free or a constant. Thus, you should be careful to use names in *ARGS* which don't occur in *BODY* (except as variable references). For example, if FOO has a macro definition of

```
(OPENLAMBDA (ENV) (FETCH (MY-RECORD-TYPE ENV) OF BAR))
```

then (FOO NIL) will expand to

```
(FETCH (MY-RECORD-TYPE NIL) OF BAR)
```

T When a macro definition is the atom T, it means that the compiler should ignore the macro, and compile the function definition; this is a simple way of turning off other macros. For example, the user may have a function that runs in both Interlisp-D and Interlisp-10, but has a macro definition that should only be used when compiling in Interlisp-10. If the MACRO property has the macro specification, a DMACRO of T will cause it to be ignored by the Interlisp-D compiler. This DMACRO would not be necessary if the macro were specified by a 10MACRO instead of a MACRO.

(**=** **.** *OTHER-FUNCTION*) A simple way to tell the compiler to compile one function exactly as it would compile another. For example, when compiling in Interlisp-D, FRPLACAs are treated as RPLACAs. This is achieved by having FRPLACA have a DMACRO of (= . RPLACA).

(**LITATOM** *EXPRESSION*) If a macro definition begins with a symbol other than those given above, this allows *computation* of the Interlisp expression to be evaluated or compiled in place of the form. LITATOM is bound to the CDR of the calling form, *EXPRESSION* is evaluated, and the result of this evaluation is evaluated or compiled in place of the form. For example, LIST could be compiled using the computed macro:

```
[X (LIST 'CONS (CAR X)(AND (CDR X)(CONS 'LIST (CDR X]
```

This would cause (LIST X Y Z) to compile as (CONS X (CONS Y (CONS Z NIL))). Note the recursion in the macro expansion.

If the result of the evaluation is the symbol IGNOREMACRO, the macro is ignored and the compilation of the expression proceeds as if there were no macro definition. If the symbol in question is normally treated specially by the compiler (CAR, CDR, COND, AND, etc.), and also has a macro, if the macro expansion returns IGNOREMACRO, the symbol will still be treated specially.

In Interlisp-10, if the result of the evaluation is the atom INSTRUCTIONS, no code will be generated by the compiler. It is then assumed the evaluation was done for effect and the necessary code, if any, has been added. This is a way of giving direct instructions to the compiler if you understand it.

It is often useful, when constructing complex macro expressions, to use the BQUOTE facility (see the Read Macros section of Chapter 25).

The following function is quite useful for debugging macro definitions:

(**EXPANDMACRO** *EXP QUIETFLG* – –) [Function]

Takes a form whose CAR has a macro definition and expands the form as it would be compiled. The result is prettyprinted, unless *QUIETFLG*=T, in which case the result is simply returned.

Note: EXPANDMACRO only works on Interlisp macros. Use CL:MACROEXPAND-1 to expand Interlisp macros visible to the Common Lisp interpreter and compliler.

DEFMACRO

Macros defined with the function DEFMACRO are much like "computed" macros (see the above section), in that they are defined with a form that is evaluated, and the result of the evaluation is used (evaluated or compiled) in place of the macro call. However, DEFMACRO macros support complex argument lists with optional arguments, default values, and keyword arguments as well as argument list destructuring.

(**DEFMACRO** *NAME ARGS FORM*) [NLambda NoSpread Function]

Defines *NAME* as a macro with the arguments *ARGS* and the definition form *FORM* (*NAME*, *ARGS*, and *FORM* are unevaluated). If an expression starting with *NAME* is evaluated or compiled, arguments are bound according to *ARGS*, *FORM* is evaluated, and the value of *FORM* is evaluated or compiled instead. The interpretation of *ARGS* is described below.

Note: Like the function DEFMACRO in Common Lisp, this function currently removes any function definition for *NAME*.

ARGS is a list that defines how the argument list passed to the macro *NAME* is interpreted. Specifically, *ARGS* defines a set of variables that are set to various arguments in the macro call (unevaluated), that *FORM* can reference to construct the macro form.

In the simplest case, *ARGS* is a simple list of variable names that are set to the corresponding elements of the macro call (unevaluated). For example, given:

```
(DEFMACRO FOO (A B) (LIST 'PLUS A B B))
```

The macro call (FOO X (BAR Y Z)) will expand to (PLUS X (BAR Y Z) (BAR Y Z)).

"&-keywords" (beginning with the character "&") that are used to set variables to particular items from the macro call form, as follows:

&OPTIONAL Used to define optional arguments, possibly with default values. Each element on *ARGS* after &OPTIONAL until the next &-keyword or the end of the list defines an optional argument, which can either be a symbol or a list, interpreted as follows:

VAR

If an optional argument is specified as a symbol, that variable is set to the corresponding element of the macro call (unevaluated).

(*VAR DEFAULT*)

If an optional argument is specified as a two element list, *VAR* is the variable to be set, and *DEFAULT* is a form that is evaluated and used as the default if there is no corresponding element in the macro call.

(*VAR DEFAULT VARSETP*)

If an optional argument is specified as a three element list, *VAR* and *DEFAULT* are the variable to be set and the default form, and *VARSETP* is a variable that is set to T if the optional argument is given in the macro call, NIL otherwise. This can be used to determine whether the argument was not given, or whether it was specified with the default value.

For example, after (DEFMACRO FOO (&OPTIONAL A (B 5) (C 6 CSET)) FORM) expanding the macro call (FOO) would cause FORM to be evaluated with A set to NIL, B set to 5, C set to 6, and CSET set to NIL. (FOO 4 5 6) would be the same, except that A would be set to 4 and CSET would be set to T.

&REST
&BODY   Used to get a list of all additional arguments from the macro call. Either &REST or &BODY should be followed by a single symbol, which is set to a list of all arguments to the macro after the position of the &-keyword. For example, given

(DEFMACRO FOO (A B &REST C) FORM)

expanding the macro call (FOO 1 2 3 4 5) would cause FORM to be evaluated with A set to 1, B set to 2, and C set to (3 4 5).

If the macro calling form contains keyword arguments (see &KEY below), these are included in the &REST list.

&KEY   Used to define keyword arguments, that are specified in the macro call by including a "keyword" (a symbol starting with the character ":") followed by a value.

Each element on ARGS after &KEY until the next &-keyword or the end of the list defines a keyword argument, which can either be a symbol or a list, interpreted as follows:

VAR
(*VAR*)
((*KEYWORD VAR*))

If a keyword argument is specified by a single symbol *VAR*, or a one-element list containing *VAR*, it is set to the value of a keyword argument, where the keyword used is created by adding the character ":" to the front of *VAR*. If a keyword argument is specified by a single-element list containing a two-element list, *KEYWORD* is interpreted as the keyword (which should start with the letter ":"), and *VAR* is the variable to set.

(*VAR DEFAULT*)
((*KEYWORD VAR*) *DEFAULT*)

```
(VAR DEFAULT VARSETP)
((KEYWORD VAR) DEFAULT VARSETP)
```

If a keyword argument is specified by a two- or three-element list, the first element of the list specifies the keyword and variable to set as above. Similar to &OPTIONAL (above), the second element *DEFAULT* is a form that is evaluated and used as the default if there is no corresponding element in the macro call, and the third element *VARSETP* is a variable that is set to T if the optional argument is given in the macro call, NIL otherwise.

For example, the form

```
(DEFMACRO FOO (&KEY A (B 5 BSET) ((:BAR C) 6 CSET)) FORM)
```

Defines a macro with keys :A, :B (defaulting to 5), and :BAR. Expanding the macro call (FOO :BAR 2 :A 1) would cause FORM to be evaluated with A set to 1, B set to 5, BSET set to NIL, C set to 2, and CSET set to T.

&ALLOW-OTHER-KEYS It is an error for any keywords to be supplied in a macro call that are not defined as keywords in the macro argument list, unless either the &-keyword &ALLOW-OTHER-KEYS appears in *ARGS*, or the keyword :ALLOW-OTHER-KEYS (with a non-NIL value) appears in the macro call.

&AUX Used to bind and initialize auxiliary varables, using a syntax similar to PROG (see the PROG and Associated Control Functions section of Chapter 9). Any elements after &AUX should be either symbols or lists, interpreted as follows:

VAR

Single symbols are interpreted as auxiliary variables that are initially bound to NIL.

(*VAR EXP*)

If an auxiliary variable is specified as a two element list, *VAR* is a variable initially bound to the result of evaluating the form *EXP*.

For example, given

```
(DEFMACRO FOO (A B &AUX C (D 5)) FORM)
```

C will be bound to NIL and D to 5 when *FORM* is evaluated.

&WHOLE Used to get the whole macro calling form. Should be the first element of *ARGS*, and should be followed by a single symbol, which is set to the entire macro calling form. Other &-keywords or arguments can follow. For example, given

```
(DEFMACRO FOO (&WHOLE X A B) FORM)
```

Expanding the macro call (FOO 1 2) would cause FORM to be evaluated with X set to (FOO 1 2), A set to 1, and B set to 2.

DEFMACRO macros also support argument list "destructuring," a facility for accessing the structure of individual arguments to a macro. Any place

in an argument list where a symbol is expected, an argument list (in the form described above) can appear instead. Such an embedded argument list is used to match the corresponding parts of that particular argument, which should be a list structure in the same form. In the simplest case, where the embedded argument list does not include &-keywords, this provides a simple way of picking apart list structures passed as arguments to a macro. For example, given

```
(DEFMACRO FOO (A (B (C . D)) E) FORM)
```

Expanding the macro call `(FOO 1 (2 (3 4 5)) 6)` would cause `FORM` to be evaluated with with `A` set to `1`, `B` set to `2`, `C` set to `3`, `D` set to `(4 5)`, and `E` set to `6`. Note that the embedded argument list `(B (C . D))` has an embedded argument list `(C . D)`. Also notice that if an argument list ends in a dotted pair, that the final symbol matches the rest of the arguments in the macro call.

An embedded argument list can also include `&`-keywords, for interpreting parts of embedded list structures as if they appeared in a top-level macro call. For example, given

```
(DEFMACRO FOO (A (B &OPTIONAL (C 6)) D) FORM)
```

Expanding the macro call `(FOO 1 (2) 3)` would cause `FORM` to be evaluated with with `A` set to `1`, `B` set to `2`, `C` set to `6` (because of the default value), and `D` set to `3`.

Warning: Embedded argument lists can only appear in positions in an argument list where a list is otherwise not accepted. In the above example, it would not be possible to specify an embedded argument list after the `&OPTIONAL` keyword, because it would be interpreted as an optional argument specification (with variable name, default value, set variable). However, it would be possible to specify an embedded argument list as the first element of an optional argument specification list, as so:

```
(DEFMACRO FOO (A (B &OPTIONAL ((X (Y) Z)
    '(1 (2) 3))) D) FORM)
```

In this case, `X`, `Y`, and `Z` default to `1`, `2`, and `3`, respectively. Note that the "default" value has to be an appropriate list structure. Also, in this case either the whole structure `(X (Y) Z)` can be supplied, or it can be defaulted (i.e., is not possible to specify `X` while letting `Y` default).

Interpreting Macros

When the interpreter encounters a form `CAR` of which is an undefined function, it tries interpreting it as a macro. If `CAR` of the form has a macro definition, the macro is expanded, and the result of this expansion is evaluated in place of the original form. `CLISPTRAN` (see the Miscellaneous Functions and Variables section of Chapter 21) is used to save the result of this expansion so that the expansion only has to be done once. On subsequent occasions, the translation (expansion) is retrieved from `CLISPARRAY` the same as for other `CLISP` constructs.

Note: Because of the way that the evaluator processes macros, if you have a macro on `FOO`, then typing `(FOO 'A 'B)` will work, but `FOO(A B)` will not work.

Medley uses "deep binding." Every time a function is entered, a basic frame containing the new variables is put on top of the stack. Therefore, any variable reference requires searching the stack for the first instance of that variable, which makes free variable use somewhat more expensive than in a shallow binding scheme. On the other hand, spaghetti stack operations are considerably faster. Some other tricks involving copying freely-referenced variables to higher frames on the stack are also used to speed up the search.

The basic frames are allocated on a stack; for most user purposes, these frames should be thought of as containing the variable names associated with the function call, and the *current* values for that frame. The descriptions of the stack functions in below are presented from this viewpoint. Both interpreted and compiled functions store both the names and values of variables so that interpreted and compiled functions are compatible and can be freely intermixed, i.e., free variables can be used with no SPECVAR declarations necessary. However, it is possible to *suppress* storing of names in compiled functions, either for efficiency or to avoid a clash, via a LOCALVAR declaration (see the Local Variables and Special Variables section of Chapter 18). The names are also very useful in debugging, for they make possible a complete symbolic backtrace in case of error.

In addition to the binding information, additional information is associated with each function call: access information indicating the path to search the basic frames for variable bindings, control information, and temporary results are also stored on the stack in a block called the frame extension. The interpreter also stores information about partially evaluated expressions as described in the Stack and Interpreter section of Chapter 11.

Spaghetti Stack

The Bobrow/Wegbreit paper, "A Model and Stack Implementation for Multiple Environments" (*Communications of the ACM,* Vol. 16, 10, October 1973.), describes an access and control mechanism more general than a simple linear stack. The access and control mechanism used by Interlisp is a slightly modified version of the one proposed by Bobrow and Wegbreit. This mechanism is called the "spaghetti stack."

The spaghetti system presents the access and control stack as a data structure composed of "frames." The functions described below operate on this structure. These primitives allow user functions to manipulate the stack in a machine independent way. Backtracking, coroutines, and more sophisticated control schemes can be easily implemented with these primitives.

The evaluation of a function requires the allocation of storage to hold the values of its local variables during the computation. In addition to variable bindings, an activation of a function requires a return link (indicating where control is to go after the completion of the computation) and room for temporaries needed during the computation. In the spaghetti system, one "stack" is used for storing all this information, but it is best to view this stack as a tree of linked objects called frame extensions (or simply frames).

A frame extension is a variable sized block of storage containing a frame name, a pointer to some variable bindings (the BLINK), and two pointers to other frame extensions (the ALINK and CLINK). In addition to these components, a frame extension contains other information (such as temporaries and reference counts) that does not interest us here.

The block of storage holding the variable bindings is called a basic frame. A basic frame is essentially an array of pairs, each of which contains a variable name and its value. The reason frame extensions point to basic frames (rather than just having them "built in") is so that two frame extensions can share a common basic frame. This allows two processes to communicate via shared variable bindings.

The chain of frame extensions which can be reached via the successive ALINKs from a given frame is called the "access chain" of the frame. The first frame in the access chain is the starting frame. The chain through successive CLINKs is called the "control chain".

A frame extension completely specifies the variable bindings and control information necessary for the evaluation of a function. Whenever a function (or in fact, any form which generally binds local variables) is evaluated, it is associated with some frame extension.

In the beginning there is precisely one frame extension in existence. This is the frame in which the top-level call to the interpreter is being run. This frame is called the "top-level" frame.

Since precisely one function is being executed at any instant, exactly one frame is distinguished as having the "control bubble" in it. This frame is called the active frame. Initially, the top-level frame is the active frame. If the computation in the active frame invokes another function, a new basic frame and frame extension are built. The frame name of this basic frame will be the name of the function being called. The ALINK, BLINK, and CLINK of the new frame all depend on precisely how the function is invoked. The new function is then run in this new frame by passing control to that frame, i.e., it is made the active frame.

Once the active computation has been completed, control normally returns to the frame pointed to by the CLINK of the active frame. That is, the frame in the CLINK becomes the active frame.

In most cases, the storage associated with the basic frame and frame extension just abandoned can be reclaimed. However, it is possible to obtain a pointer to a frame extension and to "hold on" to this frame even after it has been exited. This pointer can be used later to run another computation in that environment, or even "continue" the exited computation.

A separate data type, called a stack pointer, is used for this purpose. A stack pointer is just a cell that literally points to a frame extension. Stack pointers print as #ADR/FRAMENAME, e.g., #1,13636/COND. Stack pointers are returned by many of the stack manipulating functions described below. Except for certain abbreviations (such as "the frame with such-and-such a name"), stack pointers are the only way you can reference a frame extension. As long as you have a stack pointer which references a frame extension, that frame extension (and all those that can be reached from it) will not be garbage collected.

Two stack pointers referencing the same frame extension are *not* necessarily EQ, i.e., (EQ (STKPOS 'FOO) (STKPOS 'FOO)) = NIL. However, EQP can be used to test if two different stack pointers reference the same frame extension (see the Equality Predicates section of Chapter 9).

It is possible to evaluate a form with respect to an access chain other than the current one by using a stack pointer to refer to the head of the access chain desired. Note, however, that this can be very expensive when using a shallow binding scheme such as that in Interlisp-10. When evaluating the form, since all references to variables under the shallow binding scheme go through the variable's value cell, the values in the value cells must be adjusted to reflect the values appropriate to the desired access chain. This is done by changing all the bindings on the current access chain (all the name-value pairs) so that they contain the value current at the time of the call. Then along the new access path, all

bindings are made to contain the previous value of the variable, and the current value is placed in the value cell. For that part of the access path which is shared by the old and new chain, no work has to be done. The context switching time, i.e. the overhead in switching from the current, active, access chain to another one, is directly proportional to the size of the two branches that are not shared between the access contexts. This cost should be remembered in using generators and coroutines (see the Generators section below).

Stack Functions

In the descriptions of the stack functions below, when we refer to an argument as a stack descriptor, we mean that it is one of the following:

A stack pointer — An object that points to a frame on the stack. Stack pointers are returned by many of the stack manipulating functions described below.

NIL — Specifies the active frame; that is, the frame of the stack function itself.

T — Specifies the top-level frame.

A symbol — Specifies the first frame (along the control chain from the active frame) that has the frame name LITATOM. Equivalent to (STKPOS LITATOM -1).

A list of symbols — Specifies the first frame (along the control chain from the active frame) whose frame name is included in the list.

A number N — Specifies the Nth frame back from the active frame. If N is negative, the control chain is followed, otherwise the access chain is followed. Equivalent to (STKNTH N).

In the stack functions described below, the following errors can occur: The error Illegal stack arg occurs when a stack descriptor is expected and the supplied argument is either not a legal stack descriptor (i.e., not a stack pointer, symbol, or number), or is a symbol or number for which there is no corresponding stack frame, e.g., (STKNTH -1 'FOO) where there is no frame named FOO in the active control chain or (STKNTH -10 'EVALQT). The error Stack pointer has been released occurs whenever a released stack pointer is supplied as a stack descriptor argument for any purpose other than as a stack pointer to re-use.

Note: The creation of a single stack pointer can result in the retention of a large amount of stack space. Therefore, one should try to release stack pointers when they are no longer needed (see the Releasing and Reusing Stack Pointers section below).

In Lisp there is a fixed ammount of space allocated for the stack. When most of this space is exhausted, the STACK OVERFLOW error occurs and the debugger will be invoked. You will still have a little room on the stack to use inside the debugger. If you use up this last little bit of stack you will encounter a "hard" stack overflow. A "hard" stack overflow will put you into URaid (see the documentation on URaid).

Searching the Stack

(**STKPOS** *FRAMENAME N POS OLDPOS*)   [Function]

> Returns a stack pointer to the *N*th frame with frame name *FRAMENAME*. The search begins with (and includes) the frame specified by the stack descriptor *POS*. The search proceeds along the control chain from *POS* if *N* is negative, or along the access chain if *N* is positive. If *N* is NIL, -1 is used. Returns a stack pointer to the frame if such a frame exists, otherwise returns NIL. If *OLDPOS* is supplied and is a stack pointer, it is reused. If *OLDPOS* is supplied and is a stack pointer and STKPOS returns NIL, *OLDPOS* is released. If *OLDPOS* is not a stack pointer it is ignored.

(**STKNTH** *N POS OLDPOS*)   [Function]

> Returns a stack pointer to the *N*th frame back from the frame specified by the stack descriptor *POS*. If *N* is negative, the control chain from *POS* is followed. If *N* is positive the access chain is followed. If *N* equals 0, STKNTH returns a stack pointer to *POS* (this provides a way to copy a stack pointer). Returns NIL if there are fewer than *N* frames in the appropriate chain. If *OLDPOS* is supplied and is a stack pointer, it is reused. If *OLDPOS* is not a stack pointer it is ignored.
>
> **Note**: (STKNTH 0) causes an error, Illegal stack arg; it is not possible to create a stack pointer to the active frame.

(**STKNAME** *POS*)   [Function]

> Returns the frame name of the frame specified by the stack descriptor *POS*.

(**SETSTKNAME** *POS NAME*)   [Function]

> Changes the frame name of the frame specified by *POS* to be *NAME*. Returns *NAME*.

(**STKNTHNAME** *N POS*)   [Function]

> Returns the frame name of the *N*th frame back from *POS*. Equivalent to (STKNAME (STKNTH *N POS*)) but avoids creation of a stack pointer.

In summary, STKPOS converts function names to stack pointers, STKNTH converts numbers to stack pointers, STKNAME converts stack pointers to function names, and STKNTHNAME converts numbers to function names.

Variable Bindings in Stack Frames

The following functions are used for accessing and changing bindings. Some of functions take an argument, *N*, which specifies a particular binding in the basic frame. If *N* is a literal atom, it is assumed to be the name of a variable bound in the basic frame. If *N* is a number, it is assumed to reference the *N*th binding in the basic frame. The first binding is 1. If the basic frame contains no binding with the given name or if the number is too large or too small, the error Illegal arg occurs.

(**STKSCAN** *VAR IPOS OPOS*)   [Function]

> Searches beginning at *IPOS* for a frame in which a variable named *VAR* is bound. The search follows the access chain. Returns a stack pointer to the frame if found, otherwise returns NIL. If *OPOS* is a stack pointer it is reused, otherwise it is ignored.

(**FRAMESCAN** *ATOM POS*)   [Function]

> Returns the relative position of the binding of *ATOM* in the basic frame of *POS*. Returns NIL if *ATOM* is not found.

(**STKARG** *N POS* −)   [Function]

> Returns the value of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or number.

(**STKARGNAME** *N POS*)   [Function]

> Returns the name of the binding specified by *N*, in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or number.

(**SETSTKARG** *N POS VAL*)   [Function]

> Sets the value of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns *VAL*.

(**SETSTKARGNAME** *N POS NAME*)   [Function]

> Sets the variable name to *NAME* of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns *NAME*. This function does not work for interpreted frames.

(**STKNARGS** *POS* −)   [Function]

> Returns the number of arguments bound in the basic frame of the frame specified by the stack descriptor *POS*.

(**VARIABLES** *POS*)   [Function]

> Returns a list of the variables bound at *POS*.

(**STKARGS** *POS* −)   [Function]

> Returns a list of the values of the variables bound at *POS*.

Evaluating Expressions in Stack Frames

The following functions are used to evaluate an expression in a different environment:

(**ENVEVAL** *FORM APOS CPOS AFLG CFLG*)   [Function]

> Evaluates *FORM* in the environment specified by *APOS* and *CPOS*. That is, a new active frame is created with the frame specified by the stack descriptor *APOS* as its ALINK, and the frame specified by the stack descriptor *CPOS* as its CLINK. Then *FORM* is evaluated. If

AFLG is not NIL, and *APOS* is a stack pointer, then *APOS* will be released. Similarly, if *CFLG* is not NIL, and *CPOS* is a stack pointer, then *CPOS* will be released.

(**ENVAPPLY** *FN ARGS APOS CPOS AFLG CFLG*)   [Function]

APPLYs *FN* to *ARGS* in the environment specified by *APOS* and *CPOS*. *AFLG* and *CFLG* have the same interpretation as with ENVEVAL.

(**EVALV** *VAR POS RELFLG*)   [Function]

Evaluates *VAR*, where *VAR* is assumed to be a symbol, in the access environment specifed by the stack descriptor *POS*. If *VAR* is unbound, EVALV returns NOBIND and does not generate an error. If *RELFLG* is non-NIL and *POS* is a stack pointer, it will be released after the variable is looked up. While EVALV could be defined as (ENVEVAL *VAR POS* NIL *RELFLG*) it is in fact somewhat faster.

(**STKEVAL** *POS FORM FLG* —)   [Function]

Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*. If *FLG* is not NIL and *POS* is a stack pointer, releases *POS*. The definition of STKEVAL is (ENVEVAL *FORM POS* NIL *FLG*).

(**STKAPPLY** *POS FN ARGS FLG*)   [Function]

Like STKEVAL but applies *FN* to *ARGS*.

Altering Flow of Control

The following functions are used to alter the normal flow of control, possibly jumping to a different frame on the stack. RETEVAL and RETAPPLY allow evaluating an expression in the specified environment first.

(**RETFROM** *POS VAL FLG*)   [Function]

Return from the frame specified by the stack descriptor *POS*, with the value *VAL*. If *FLG* is not NIL, and *POS* is a stack pointer, then *POS* is released. An attempt to RETFROM the top level (e.g., (RETFROM T)) causes an error, Illegal stack arg. RETFROM can be written in terms of ENVEVAL as follows:

```
(RETFROM
    (LAMBDA (POS VAL FLG)
       (ENVEVAL (LIST 'QUOTE VAL)
               NIL
               (if (STKNTH -1 POS
                   (if FLG then POS))
               else (ERRORX (LIST 19 POS)))
               NIL
               T)))
```

(**RETTO** *POS VAL FLG*)   [Function]

Like RETFROM, but returns *to* the frame specified by *POS*.

(**RETEVAL** *POS FORM FLG* —) [Function]

> Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*, and then returns from *POS* with that value. If *FLG* is not NIL and *POS* is a stack pointer, then *POS* is released. The definition of RETEVAL is equivalent to (ENVEVAL *FORM POS* (STKNTH -1 *POS*) *FLG* T), but RETEVAL does not create a stack pointer.

(**RETAPPLY** *POS FN ARGS FLG*) [Function]

> Like RETEVAL but applies *FN* to *ARGS*.

Releasing and Reusing Stack Pointers

The following functions and variables are used for manipulating stack pointers:

(**STACKP** *X*) [Function]

> Returns *X* if *X* is a stack pointer, otherwise returns NIL.

(**RELSTK** *POS*) [Function]

> Release the stack pointer *POS* (see below). If *POS* is not a stack pointer, does nothing. Returns *POS*.

(**RELSTKP** *X*) [Function]

> Returns T is *X* is a released stack pointer, NIL otherwise.

(**CLEARSTK** *FLG*) [Function]

> If *FLG* is T, returns a list of all the active (unreleased) stack pointers. If *FLG* is NIL, this call is a no-op. The abillity to clear all stack pointers is inconsistent with the modularity implicit in a multi processing environment.

CLEARSTKLST [Variable]

> A variable used by the top-level executive. Every time the top-level executive is re-entered (e.g., following errors, or Control-D), CLEARSTKLST is checked. If its value is T, all active stack pointers are released using CLEARSTK. If its value is a list, then all stack pointers on that list are released. If its value is NIL, nothing is released. CLEARSTKLST is initially T.

NOCLEARSTKLST [Variable]

> A variable used by the top-level executive. If CLEARSTKLST is T (see above) all active stack pointers *except* those on NOCLEARSTKLST are released. NOCLEARSTKLST is initially NIL.

Creating a single stack pointer can cause the retention of a large amount of stack space. Furthermore, this space will not be freed until the next garbage collection, *even if the stack pointer is no longer being used*, unless the stack pointer is explicitly released or reused. If there is sufficient amount of stack space tied up in this fashion, a STACK OVERFLOW condition can occur, even in the simplest of computations. For this reason, you should consider releasing a stack pointer when the environment referenced by the stack pointer is no longer needed.

The effects of releasing a stack pointer are:

1. The link between the stack pointer and the stack is broken by setting the contents of the stack pointer to the "released mark". A released stack pointer prints as #*ADR*/#*0*.

2. If this stack pointer was the last remaining reference to a frame extension; that is, if no other stack pointer references the frame extension and the extension is not contained in the active control or access chain, then the extension may be reclaimed, and is reclaimed immediately. The process repeats for the access and control chains of the reclaimed extension so that all stack space that was reachable only from the released stack pointer is reclaimed.

A stack pointer may be released using the function RELSTK, but there are some cases for which RELSTK is not sufficient. For example, if a function contains a call to RETFROM in which a stack pointer was used to specify where to return to, it would not be possible to simultaneously release the stack pointer. (A RELSTK appearing in the function following the call to RETFROM would not be executed!) To permit release of a stack pointer in this situation, the stack functions that relinquish control have optional flag arguments to denote whether or not a stack pointer is to be released (*AFLG* and *CFLG*). Note that in this case releasing the stack pointer will *not* cause the stack space to be reclaimed immediately because the frame referenced by the stack pointer will have become part of the active environment.

Another way to avoid creating new stack pointers is to *reuse* stack pointers that are no longer needed. The stack functions that create stack pointers (STKPOS, STKNTH, and STKSCAN) have an optional argument that is a stack pointer to reuse. When a stack pointer is reused, two things happen. First the stack pointer is released (see above). Then the pointer to the new frame extension is deposited in the stack pointer. The old stack pointer (with its new contents) is returned as the value of the function. Note that the reused stack pointer will be released even if the function does not find the specified frame.

Even if stack pointers are explicitly being released, *creating* many stack pointers can cause a garbage collection of stack pointer space. Thus, if your application requires creating many stack pointers, you definitely should take advantage of reusing stack pointers.

Backtrace Functions

The following functions perform a "backtrace," printing information about every frame on the stack. Arguments allow only backtracing a selected range of the stack, skipping selected frames, and printing different amounts of information about each frame.

> (**BACKTRACE** *IPOS EPOS FLAGS FILE PRINTFN*)   [Function]
>
> > Performs a backtrace beginning at the frame specified by the stack descriptor *IPOS*, and ending with the frame specified by the stack descriptor *EPOS*. *FLAGS* is a number in which the options of the BACKTRACE are encoded. If a bit is set, the corresponding information is included in the backtrace.
> >
> > 1Q - print arguments of non-SUBRs
> >
> > 2Q - print temporaries of the interpreter
> >
> > 4Q - print SUBR arguments and local variables
> >
> > 10Q - omit printing of UNTRACE: and function names
> >
> > 20Q - follow access chain instead of control chain

40Q - print temporaries, i.e. the blips (see the stack and interpreter section below)

For example: If *FLAGS* = 47Q, everything is printed. If *FLAGS* = 21Q, follows the access chain, prints arguments.

FILE is the file that the backtrace is printed to. *FILE* must be open. *PRINTFN* is used when printing the values of variables, temporaries, blips, etc. *PRINTFN* = NIL defaults to PRINT.

(**BAKTRACE** *IPOS EPOS SKIPFNS FLAGS FILE*)   [Function]

Prints a backtrace from *IPOS* to *EPOS* onto *FILE*. *FLAGS* specifies the options of the backtrace, e.g., do/don't print arguments, do/don't print temporaries of the interpreter, etc., and is the same as for BACKTRACE.

SKIPFNS is a list of functions. As BAKTRACE scans down the stack, the stack name of each frame is passed to each function in *SKIPFNS*, and if any of them returnS non-NIL, *POS* is skipped (including all variables).

BAKTRACE collapses the sequence of several function calls corresponding to a call to a system package into a single "function" using BAKTRACELST as described below. For example, any call to the editor is printed as **EDITOR**, a break is printed as **BREAK**, etc.

BAKTRACE is used by the BT, BTV, BTV+, BTV*, and BTV! break commands, with *FLAGS* = 0, 1, 5, 7, and 47Q respectively.

If SYSPRETTYFLG = T, the values arguments and local variables will be prettyprinted.

BAKTRACELST   [Variable]

Used to tell BAKTRACE (therefore, the BT, BTV, etc. commands) to abbreviate various sequences of function calls on the stack by a single key, e.g. **BREAK**, **EDITOR**, etc.

Each entry on BAKTRACELST is a list of the form (*FRAMENAME KEY . PATTERN*) or (*FRAMENAME* (*KEY$_1$. PATTERN$_1$*) ... (*KEY$_N$. PATTERN$_N$*)), where a pattern is a list of elements that are either atoms, which match a single frame, or lists, which are interpreted as a list of alternative patterns, e.g. (PROGN **BREAK** EVAL ((ERRORSET BREAK1A BREAK1) (BREAK1)))

BAKTRACE operates by scanning up the stack and, at each point, comparing the current frame name, with the frame names on BAKTRACELST, i.e. it does an ASSOC. If the frame name does appear, BAKTRACE attempts to match the stack as of that point with (one of) the patterns. If the match is successful, BAKTRACE prints the corresponding key, and continues with where the match left off. If the frame name does not appear, or the match fails, BAKTRACE simply prints the frame name and continues with the next higher frame (unless the *SKIPFNS* applied to the frame name are non-NIL as described above).

Matching is performed by comparing symbols in the pattern with the current frame name, and matching lists as patterns, i.e. sequences of function calls, always working up the stack. For example, either of the sequence of function calls "... BREAK1 BREAK1A ERRORSET EVAL PROGN ..."

or "`... BREAK1 EVAL PROGN ...`" would match with the sample entry given above, causing `**BREAK**` to be printed.

Special features:

- The symbol `&` can be used to match any frame.

- The pattern "`-`" can be used to match nothing. `-` is useful for specifying an optional match, e.g. the example above could also have been written as (`PROGN **BREAK** EVAL ((ERRORSET BREAK1A) -) BREAK1`).

- It is not necessary to provide in the pattern for matching dummy frames, i.e. frames for which `DUMMYFRAMEP` (see below) is true. When working on a match, the matcher automatically skips over these frames when they do not match.

- If a match succeeds and the *KEY* is `NIL`, nothing is printed. For example, (`*PROG*LAM NIL EVALA *ENV`). This sequence will occur following an error which then causes a break if some of the function's arguments are `LOCALVARS`.

Other Stack Functions

(**DUMMYFRAMEP** *POS*)   [Function]

> Returns `T` if you never wrote a call to the function at *POS*, e.g. in Interlisp-10, `DUMMYFRAMEP` is `T` for `*PROG*LAM`, `*ENV*`, and `FOOBLOCK` frames (see the Block Compiling section of Chapter 18).

`REALFRAMEP` and `REALSTKNTH` can be used to write functions which manipulate the stack and work on either interpreted or compiled code:

(**REALFRAMEP** *POS INTERPFLG*)   [Function]

> Returns *POS* if *POS* is a "real" frame, i.e. if *POS* is not a dummy frame and *POS* is a frame that does not disappear when compiled (such as `COND`); otherwise `NIL`. If *INTERPFLG* = `T`, returns `T` if *POS* is not a dummy frame. For example, if (`STKNAME POS`) = `COND`, (`REALFRAMEP POS`) is `NIL`, but (`REALFRAMEP POS T`) is `T`.

(**REALSTKNTH** *N POS INTERPFLG OLDPOS*)   [Function]

> Returns a stack pointer to the *N*th (or –*N*th) frames for which (`REALFRAMEP POS INTERPFLG`) is *POS*.

(**MAPDL** *MAPDLFN MAPDLPOS*)   [Function]

> Starts at *MAPDLPOS* and applies the function *MAPDLFN* to two arguments (the frame name and a stack pointer to the frame), for each frame until the top of the stack is reached. Returns `NIL`. For example,

```
[MAPDL (FUNCTION (LAMBDA (X POS)
    (if (IGREATERP (STKNARGS POS) 2) then (PRINT X)]
```

> will print all functions of more than two arguments.

(**SEARCHPDL** *SRCHFN SRCHPOS*)   [Function]

> Like MAPDL, but searches the stack starting at position *SRCHPOS* until it finds a frame for which *SRCHFN*, a function of two arguments applied to the *name* of the frame and the frame itself, is not NIL. Returns (*NAME* . *FRAME*) if such a frame is found, otherwise NIL.

The Stack and the Interpreter

In addition to the names and values of arguments for functions, information regarding partially-evaluated expressions is kept on the push-down list. For example, consider the following definition of the function FACT (intentionally faulty):

```
(FACT
    [LAMBDA (N)
        (COND
          ((ZEROP N)
            L)
          (T (ITIMES N (FACT (SUB1 N])
```

In evaluating the form (FACT 1), as soon as FACT is entered, the interpreter begins evaluating the implicit PROGN following the LAMBDA. The first function entered in this process is COND. COND begins to process its list of clauses. After calling ZEROP and getting a NIL value, COND proceeds to the next clause and evaluates T. Since T is true, the evaluation of the implicit PROGN that is the consequent of the T clause is begun. This requires calling the function ITIMES. However before ITIMES can be called, its arguments must be evaluated. The first argument is evaluated by retrieving the current binding of N from its value cell; the second involves a recursive call to FACT, and another implicit PROGN, etc.

At each stage of this process, some portion of an expression has been evaluated, and another is awaiting evaluation. The output below (from Interlisp-10) illustrates this by showing the state of the push-down list at the point in the computation of (FACT 1) when the unbound atom L is reached.

```
 ← FACT(1)
  u.b.a. L {in FACT} in ((ZEROP NO L)
  (L broken)
  :BTV!
     *TAIL* (L)
     *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
  COND
     *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
     *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
     N 0
  FACT
     *FORM* (FACT (SUB1 N))
     *FN* ITIMES
     *TAIL* ((FACT (SUB1 N)))
     *ARGVAL* 1
     *FORM* (ITIMES N (FACT (SUB1 N)))
     *TAIL* ((ITIMES N (FACT (SUB1 N))))
     *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
  COND
     *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
     *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))
     N 1
  FACT
```

```
**TOP**
```

Internal calls to EVAL, e.g., from COND and the interpreter, are marked on the push-down list by a special mark or blip which the backtrace prints as *FORM*. The genealogy of *FORM*'s is thus a history of the computation. Other temporary information stored on the stack by the interpreter includes the tail of a partially evaluated implicit PROGN (e.g., a cond clause or lambda expression) and the tail of a partially evaluated form (i.e., those arguments not yet evaluated), both indicated on the backtrace by *TAIL*, the values of arguments that have already been evaluated, indicated by *ARGVAL*, and the names of functions waiting to be called, indicated by *FN*. *ARG1, ..., *ARGn are used by the backtrace to indicate the (unnamed) arguments to SUBRs.

Note that a function is not actually entered and does not appear on the stack, until its arguments have been evaluated (except for nlambda functions, of course). Also note that the *ARG1, *FORM*, *TAIL*, etc. "bindings" comprise the actual working storage. In other words, in the above example, if a (lower) function changed the value of the *ARG1 binding, the COND would continue interpreting the new binding as a list of COND clauses. Similarly, if the *ARGVAL* binding were changed, the new value would be given to ITIMES as its first argument after its second argument had been evaluated, and ITIMES was actually called.

FORM, *TAIL*, *ARGVAL*, etc., do not actually appear as variables on the stack, i.e., evaluating *FORM* or calling STKSCAN to search for it will not work. However, the functions BLIPVAL, SETBLIPVAL, and BLIPSCAN described below are available for accessing these internal blips. These functions currently know about four different types of blips:

> *FN* The name of a function about to be called

> *ARGVAL* An argument for a function about to be called

> *FORM* A form in the process of evaluation

> *TAIL* The tail of a COND clause, implicit PROGN, PROG, etc.

(**BLIPVAL** *BLIPTYP IPOS FLG*)   [Function]

> Returns the value of the specified blip of type *BLIPTYP*. If *FLG* is a number N, finds the Nth blip of the desired type, searching the control chain beginning at the frame specified by the stack descriptor *IPOS*. If *FLG* is NIL, 1 is used. If *FLG* is T, returns the number of blips of the specified type at *IPOS*.

(**SETBLIPVAL** *BLIPTYP IPOS N VAL*)   [Function]

> Sets the value of the specified blip of type *BLIPTYP*. Searches for the Nth blip of the desired type, beginning with the frame specified by the stack descriptor *IPOS*, and following the control chain.

(**BLIPSCAN** *BLIPTYP IPOS*)   [Function]

> Returns a stack pointer to the frame in which a blip of type *BLIPTYP* is located. Search begins at the frame specified by the stack descriptor *IPOS* and follows the control chain.

Generators

A *generator* is like a subroutine except that it retains information about previous times it has been called. Some of this state may be data (for example, the seed in a random number generator), and some may be in program state (as in a recursive generator which finds all the atoms in a list structure). For example, if LISTGEN is defined by:

```
(DEFINEQ (LISTGEN (L)
        (if L then (PRODUCE (CAR L))
                   (LISTGEN (CDR L)))))
```

we can use the function GENERATOR (described below) to create a generator that uses LISTGEN to produce the elements of a list one at a time, e.g.,

```
(SETQ GR (GENERATOR (LISTGEN '(A B C))))
```

creates a generator, which can be called by

```
(GENERATE GR)
```

to produce as values on successive calls, A, B, C. When GENERATE (not GENERATOR) is called the first time, it simply starts evaluating (LISTGEN '(A B C)). PRODUCE gets called from LISTGEN, and pops back up to GENERATE with the indicated value after saving the state. When GENERATE gets called again, it continues from where the last PRODUCE left off. This process continues until finally LISTGEN completes and returns a value (it doesn't matter what it is). GENERATE then returns GR itself as its value, so that the program that called GENERATE can tell that it is finished, i.e., there are no more values to be generated.

(**GENERATOR** *FORM COMVAR*)   [NLambda Function]

> An nlambda function that creates a generator which uses *FORM* to compute values. GENERATOR returns a *generator handle* which is represented by a dotted pair of stack pointers.

> *COMVAR* is optional. If its value (EVAL of) is a generator handle, the list structure and stack pointers will be reused. Otherwise, a new generator handle will be constructed.

> GENERATOR compiles open.

(**PRODUCE** *VAL*)   [Function]

> Used from within a generator to return *VAL* as the value of the corresponding call to GENERATE.

(**GENERATE** *HANDLE VAL*)   [Function]

> Restarts the generator represented by *HANDLE*. *VAL* is returned as the value of the PRODUCE which last suspended the operation of the generator. When the generator runs out of values, GENERATE returns *HANDLE* itself.

Examples:

The following function will go down recursively through a list structure and produce the atoms in the list structure one at a time.

```
(DEFINEQ (LEAVESG (L)
        (if (ATOM L)
```

```
                         then (PRODUCE L)
                         else (LEAVESG (CAR L))
                              (if (CDR L)
                               then (LEAVESG (CDR L)]
```

The following function prints each of these atoms as it appears. It illustrates how a loop can be set up to use a generator.

```
        (DEFINEQ (PLEAVESG1 (L)
              (PROG (X LHANDLE)
                (SETQ LHANDLE (GENERATOR (LEAVESG L)))
               LP (SETQ X (GENERATE LHANDLE))
                 (if (EQ X LHANDLE)
                  then (RETURN NIL))
                   (PRINT X)
                      (GO LP))]
```

The loop terminates when the value of the generator is EQ to the dotted pair which is the value produced by the call to GENERATOR. A CLISP iterative operator, OUTOF, is provided which makes it much easier to write the loop in PLEAVESG1. OUTOF (or outof) can precede a form which is to be used as a generator. On each iteration, the iteration variable will be set to successive values returned by the generator; the loop will be terminated automatically when the generator runs out. Therefore, the following is equivalent to the above program PLEAVESG1:

```
(DEFINEQ (PLEAVESG2 (L) (for X outof (LEAVESG L) do (PRINT X))]
```

Here is another example; the following form will print the first N atoms.

```
(for X outof (MAPATOMS (FUNCTION PRODUCE)) as I from 1 to N do (PRINT X))
```

Coroutines

This package provides facilities for the creation and use of fully general coroutine structures. It uses a stack pointer to preserve the state of a coroutine, and allows arbitrary switching between *N* different coroutines, rather than just a call to a generator and return. This package is slightly more efficient than the generator package described above, and allows more flexibility on specification of what to do when a coroutine terminates.

(**COROUTINE** *CALLPTR COROUTPTR COROUTFORM ENDFORM*)   [NLambda Function]

> This nlambda function is used to create a coroutine and initialize the linkage. *CALLPTR* and *COROUTPTR* are the names of two variables, which will be set to appropriate stack pointers. If the values of *CALLPTR* or *COROUTPTR* are already stack pointers, the stack pointers will be reused. *COROUTFORM* is the form which is evaluated to start the coroutine; *ENDFORM* is a form to be evaluated if *COROUTFORM* actually returns when it runs out of values.
>
> COROUTINE compiles open.

(**RESUME** *FROMPTR TOPTR VAL*)   [Function]

> Used to transfer control from one coroutine to another. *FROMPTR* should be the stack pointer for the current coroutine, which will be smashed to preserve the current state. *TOPTR* should be the stack pointer which has preserved the state of the coroutine to be transferred to, and *VAL* is the value that is to be returned to the latter coroutine as the value of the RESUME which suspended the operation of that coroutine.

For example, the following is the way one might write the LEAVES program using the coroutine package:

```
(DEFINEQ (LEAVESC (L COROUTPTR CALLPTR)
   (if (ATOM L)
    then (RESUME COROUTPTR CALLPTR L)
    else (LEAVESC (CAR L) COROUTPTR CALLPTR)
         (if (CDR L) then (LEAVESC (CDR L) COROUTPTR CALLPTR)))))]
```

A function PLEAVESC which uses LEAVESC can be defined as follows:

```
(DEFINEQ (PLEAVESC (L)
   (bind PLHANDLE LHANDLE
    first (COROUTINE PLHANDLE LHANDLE
                      (LEAVESC L LHANDLE PLHANDLE)
                      (RETFROM 'PLEAVESC))
      do (PRINT (RESUME PLHANDLE LHANDLE)))))]
```

By RESUMEing LEAVESC repeatedly, this function will print all the leaves of list L and then return out of PLEAVESC via the RETFROM. The RETFROM is necessary to break out of the non-terminating do-loop. This was done to illustrate the additional flexibility allowed through the use of *ENDFORM*.

We use two coroutines working on two trees in the example EQLEAVES, defined below. EQLEAVES tests to see whether two trees have the same leaf set in the same order, e.g., (EQLEAVES '(A B C) '(A B (C))) is true.

```
(DEFINEQ (EQLEAVES (L1 L2)
   (bind LHANDLE1 LHANDLE2 PE EL1 EL2
    first (COROUTINE PE LHANDLE1 (LEAVESC L1 LHANDLE1 PE) 'NO-MORE)
          (COROUTINE PE LHANDLE2 (LEAVESC L2 LHANDLE2 PE) 'NO-MORE)
      do (SETQ EL1 (RESUME PE LHANDLE1))
         (SETQ EL2 (RESUME PE LHANDLE2))
         (if (NEQ EL1 EL2)
          then (RETURN NIL))
    repeatuntil (EQ EL1 'NO-MORE)
    finally (RETURN T)))]
```

Possibilities Lists

A possibilities list is the interface between a generator and a consumer. The possibilities list is initialized by a call to POSSIBILITIES, and elements are obtained from it by using TRYNEXT. By using the spaghetti stack to maintain separate environments, this package allows a regime in which a generator can put a few items in a possibilities list, suspend itself until they have been consumed, and be subsequently aroused and generate some more.

(**POSSIBILITIES** *FORM*)   [NLambda Function]

> This nlambda function is used for the initial creation of a possibilities list. *FORM* will be evaluated to create the list. It should use the functions NOTE and AU-REVOIR described below to generate possibilities. Normally, one would set some variable to the possibilities list which is returned, so it can be used later, e.g.:

>   (SETQ PLIST (POSSIBILITIES (GENERFN V1 V2))).

> POSSIBILITIES compiles open.

(**NOTE** *VAL LSTFLG*)   [Function]

    Used within a generator to put items on the possibilities list being generated. If *LSTFLG* is equal to NIL, *VAL* is treated as a single item. If *LSTFLG* is non-NIL, then the list *VAL* is NCONCed on the end of the possibilities list. Note that it is perfectly reasonable to create a possibilities list using a second generator, and NOTE that list as possibilities for the current generator with *LSTFLG* equal to T. The lower generator will be resumed at the appropriate point.

(**AU-REVOIR** *VAL*)   [NoSpread Function]

    Puts *VAL* on the possibilities list if it is given, and then suspends the generator and returns to the consumer in such a fashion that control will return to the generator at the AU-REVOIR if the consumer exhausts the possibilities list.

    NIL is not put on the possibilities list unless it is explicitly given as an argument to AU-REVOIR, i.e., (AU-REVOIR) and (AU-REVOIR NIL) are *not* the same. AU-REVOIR and ADIEU are lambda nospreads to enable them to distinguish these two cases.

(**ADIEU** *VAL*)   [NoSpread Function]

    Like AU-REVOIR but releases the generator instead of suspending it.

(**TRYNEXT** *PLST ENDFORM VAL*)   [NLambda Function]

    This nlambda function allows a consumer to use a possibilities list. It removes the first item from the possibilities list named by *PLST* (i.e. *PLST* must be an atom whose value is a possiblities list), and returns that item, provided it is not a generator handle. If a generator handle is encountered, the generator is reawakened. When it returns a possibilities list, this list is added to the front of the current list. When a call to TRYNEXT causes a generator to be awakened, *VAL* is returned as the value of the AU-REVOIR which put that generator to sleep. If *PLST* is empty, it evaluates *ENDFORM* in the caller's environment.

    TRYNEXT compiles open.

(**CLEANPOSLST** *PLST*)   [Function]

    This function is provided to release any stack pointers which may be left in the *PLST* which was not used to exhaustion.

For example, FIB is a generator for fibonnaci numbers. It starts out by NOTEing its two arguments, then suspends itself. Thereafter, on being re-awakened, it will NOTE two more terms in the series and suspends again. PRINTFIB uses FIB to print the first N fibonacci numbers.

```
(DEFINEQ (FIB (F1 F2)
   (do (NOTE F1)
       (NOTE F2)
       (SETQ F1 (IPLUS F1 F2))
       (SETQ F2 (IPLUS F1 F2))
       (AU-REVOIR)]
```

Note that this AU-REVOIR just suspends the generator and adds nothing to the possibilities list except the generator.

```
(DEFINEQ (PRINTFIB (N)
    (PROG ((FL (POSSIBILITIES (FIB 0 1))))
          (RPTQ N (PRINT  (TRYNEXT FL)))
          (CLEANPOSLST FL)]
```

Note that `FIB` itself will never terminate.

[This page intentionally left blank]

Greeting and Initialization Files

Many of the features of Medley are controlled by variables that you can adjust to your own taste. In addition, you can modify the action of system functions in ways not specifically provided for by using ADVISE (see the Advise Functions section of Chapter 15). To encourage customizing Medley's environment, it includes a facility for automatically loading initialization files (or "init files") when it is first started. Each user can have a separate "user init file" that customizes Medley's environment to his/her tastes. In addition, there can be a "site init file" that applies to all users at a given physical site, setting system variables that are the same for all users such as the name of the nearest printer, etc.

The process of loading init files, also known as "greeting", occurs when a Medley system created by MAKESYS (see the Saving Virtual Memory State section below) is started for the first time. The user can also explicitly invoke the greeting operation at any time via the function GREET (below). The process of greeting includes the following steps:

1. Any previous greeting operation is undone. The side effects of the greeting operation are stored on a global variable as well as on the history list, thus enabling the previous greeting to be undone even if it has dropped off of the bottom of the history list.

2. All of the items on the list PREGREETFORMS are evaluated.

3. The site init file is loaded. GREET looks for a file by the name {DSK}INIT.LISP. If this is found, it is loaded. If it is not found, the system prints Please enter name of system init file (e.g. {server}<directory>INIT.extension): and waits for you to type a file name, followed by a carriage return. If you just type a carriage return without typing a file name, no site init file is loaded. **Note**: The site init file is loaded with *LDFLG* set to SYSLOAD, so that no file package information is saved, and nothing is printed out.

4. The user init file is loaded. The user init file is found by using the variable USERGREETFILES (described below), which is normally set in the site init file. The user init file is loaded with normal file package settings, but under errorset protection and with PRETTYHEADER set to NIL to suppress the File created message.

5. All of the items on the list POSTGREETFORMS are evaluated.

6. The greeting "Hello, *XXX*." is printed, where *XXX* is the value of the variable FIRSTNAME (if non-NIL). The variable GREETDATES (below) can be set to modify this greeting for particular dates.

    (**GREET** *NAME* —)   [Function]

        Performs the greeting for person whose username is *NAME* (if *NAME* = NIL, uses the login name). When Medley first starts up, it performs (GREET).

(**GREETFILENAME** *USER*)   [Function]

>   If *USER* is T, GREETFILENAME returns the file name of the site init file. If the file name
>   doesn't exist, you are prompted for it. Otherwise, *USER* is interpreted to be a user's
>   system name, and GREETFILENAME returns the file name for the user init file (if it exists).

USERGREETFILES   [Variable]

>   USERGREETFILES specifies a series of file names to try as the user init file. The value of
>   USERGREETFILES is a list, where each element is a list of symbols. For each item in
>   USERGREETFILES, the user name is substituted for the symbol USER and the value of
>   COMPILE.EXT (see the Cimpiler Functions section of Chapter 18) is substituted for the
>   symbol COM, and the symbols are packed into a single file name. The first such file that is
>   found is the user init file.
>
>   For example, suppose that the value of USERGREETFILES was

```
(({ERIS}< USER >LISP>INIT. COM)
 ({ERIS}< USER >LISP>INIT)
 ({ERIS}< USER >INIT. COM)
 ({ERIS}< USER >INIT))
```

>   If the user name was JONES, and the value of COMPILE.EXT was DCOM, then this would
>   search for the files {ERIS}<JONES>LISP>INIT.DCOM, {ERIS}<JONES>LISP>INIT,
>   {ERIS}<JONES>INIT.DCOM, and {ERIS}<JONES>INIT.
>
>   **Note**: The file name "specifications" in USERGREETFILES should be fully qualified,
>   including all host and directory information. The directory search path (the value of
>   DIRECTORIES, see the Searching File Directories section of Chapter 24) is *not* used to find
>   the user greet files.

GREETDATES   [Variable]

>   The value of GREETDATES can be used to specify special greeting messages for various
>   dates. GREETDATES is a list of elements of the form (*DATESTRING . STRING*), e.g.
>   ("25-DEC" . "Merry Christmas"). The user can add entries to this list in his/her
>   INIT.LISP file by using a ADDVARS file package command like (ADDVARS
>   (GREETDATES ("8-FEB" . "Happy Birthday"))). On the specified date, the
>   GREET will use the indicated salutation.
>
>   It is impossible to give a complete list of all of the variables and functions you may want
>   to set in your init files. The default values for system variables are chosen in the hope that
>   they will be correct for the majority of users, so many users get along with very small init
>   files. The following describes some of the variables that users may want to reset in their
>   init files:

>   Directories The variables DIRECTORIES and LISPUSERSDIRECTORIES (see the
>   Searching File Directories section of Chapter 24) contain lists of directories
>   used when searching for files. LOGINHOST/DIR (see the Incomplete File
>   Names section of Chapter 24) determines the default directory used when
>   you call CONN with no argument.

Fonts and Printing   The variables `DISPLAYFONTDIRECTORIES`, `DISPLAYFONTEXTENSIONS`, `INTERPRESSFONTDIRECTORIES`, and `PRESSFONTWIDTHSFILES` (see the Font Files and Font Directories section of Chapter 27) must be set before fonts can be automatically loaded from files. `DEFAULTPRINTINGHOST` (see Chapter 29) should be set before attempting to generate hardcopy to a printer.

Network Systems   `CH.DEFAULT.ORGANIZATION` and `CH.DEFAULT.DOMAIN` (see the Name and Address Conventions section of Chapter 31) should be set to the default NS organization and domain, when using NS network communications. If `CH.NET.HINT` (see the Clearinghouse Functions section of Chapter 31) is set, it can reduce the amount of time spent searching for a clearinghouse.

Medley Executive   The variable `PROMPT#FLG` (see the Changing the Programmer's Assistant section of Chapter 13) determines whether an "event number" is printed at the beginning of every input line. The function `CHANGESLICE` (see the Changing the Programmer's Assistant section of Chapter 13) can be used to change the number of events that are remembered on the history list.

Copyright Notices   `COPYRIGHTFLG`, `COPYRIGHTOWNERS`, and `DEFAULTCOPYRIGHTOWNER` (see the Copyright Notices section of Chapter 17) control the inclusion of copyright notices on source files.

Printing Functions   `**COMMENT**FLG` (see the Comment Feature section of Chapter 26) determines how program comments are printed. `FIRSTCOL`, `PRETTYFLG`, and `CLISPIFYPRETTYFLG` (see the Special Prettyprint Controls section of Chapter 26) are among the many variables controlling how functions are pretty printed.

List Structure Editor   The variable `INITIALSLST` (see the Time Stamps section of Chapter 16) is used when "time-stamps" are inserted in a function when it is edited. `EDITCHARACTERS` (see the Time Stamps section of Chapter 16) is used to set the read macros used in the teletype editor.

Idle Mode

The Medley environment runs on small single-user computers, usually located in users' offices. Often, users leave their computers up and running for days, which can cause several problems. First, the phosphor in the video display screen can be permanently marked if the same pattern is displayed for a long time (weeks). Second, if you go away, leaving a Medley system running, another person could possibly walk up and use the environment, taking advantage of any passwords that have been entered. To solve these problems, Medley implements the concept of "idle mode."

If no keyboard or mouse action has occurred for a specified time, Medley automatically enters idle mode. While idle mode is on, the display screen is blacked out, to protect the phosphor. Idle mode also runs a program to display some moving pattern on the black screen, so the screen does not appear to be broken. Usually, idle mode can be exited by pressing any key on the keyboard or mouse. However, you can optionally specify that idle mode should erase the current password cache when it is entered., and require the next user to supply a password to exit idle mode.

If either shift key is pressed while Medley is in idle mode, the current user name and the amount of time spent idling are displayed in the prompt window while the key is depressed.

Idle mode can also be entered by calling the function `IDLE` , or by selecting the Idle menu command from the background menu (see Chapter 28). The Idle menu command has subitems that allow you to interactively set the idle options (display program, erasing password, etc.) specified by the variable `IDLE.PROFILE`.

`IDLE.PROFILE` [Variable]

The value of this variable is a property list (see Chapter 3) which controls most aspects of idle mode. The following properties are recognized:

TIMEOUT   Value is a number that determines how long (in minutes) Medley will wait before automatically entering idle mode. If `NIL`, idle mode will never be entered automatically. Default is 10 minutes.

FORGET   If this is the symbol `FIRST`, your password will be erased when idle mode is entered. If non-`NIL`, your password will be erased when idle mode is exited. Initial value is `T` (erase password on exit).

If the password is erased on entry to idle mode (value `FIRST`), any programs left running when idle mode is entered will fail if they try doing anything requiring passwords (such as accessing file servers).

ALLOWED.LOGINS   The value of this property can either be a list or a non-list. If the value is `NIL` or any other non-list, idle mode is exited without requesting login.

If the value is a list the members of the list should be interpreted as follows:

*   If the value is a list containing * as it's element, login is required but anyone can exit idle mode. This will overwrite the previous user's user name and password each time idle mode is exited.

T   Let the previous user (as determined by `USERNAME`) exit idle mode. If the username has not been set, this is equivalent to *

user name   Let this specific user exit idle mode.

group name   Let any member of this group (an NS clearinghouse group) exit idle mode.

AUTHENTICATE   The value of this property determines the method used for logging in. The value can be one of the following:

T or NS   Use the NS authentication protocol. This requires that you have an NS authentication server accessible on your net.

GV   Authenticate the login via the GrapeVine protocol.

UNIX   Use the unix login mechanism.

Note: Unix is case sensitive. If you try to login but fail, you may have typed the password with the caps-lock on.

LOGIN.TIMEOUT   This is the number of seconds idle will wait for a login before resuming idle mode again.

DISPLAYFN   The value of this property, which should be a function name or lambda expression, is called to display a moving pattern on the screen while in idle mode. This function is called with one argument, a window covering the whole screen. The default is IDLE.BOUNCING.BOX (below).

Any function used as a DISPLAYFN should call BLOCK (see Chapter 23) frequently, so other programs can run during idle mode.

SAVEVM   Value is a number that determines how long (in minutes) after idle mode is entered that SAVEVM will be called to save the virtual memory. If NIL, SAVEVM is never called automatically from idle mode. Default is 10 minutes.

SUSPEND.PROCESS.NAMES   Value is a list of names. For each name on this list, if a process by that name is found, it will be suspended upon entering idle mode and woken upon exiting idle mode.

IDLE.RESETVARS   [Variable]

The value of this variable is a list of two-element lists: $((VAR_1 \; EXP_1)(VAR_2 \; EXP_2)...)$. On entering idle mode, each variable VAR_N is bound to the value of the corresponding expression EXP_N. When idle mode is exited, each variable VAR_N is reset to its original value.

IDLE.SUSPEND.PROCESS.NAMES   [Variable]

Value is a list of names. For each name on this list, if a process by that name is found, it will be suspended upon entering idle mode and woken upon exiting idle mode.

IDLE.PROFILE   [Variable]

The value of this variable determines the menu raised by selecting the Display subitem of the Idle background menu command. It should be in the format used for the ITEMS field of a menu (see Chapter 28), with the selection of an item returning the appropriate display function.

(**IDLE.BOUNCING.BOX** *WINDOW BOX WAIT*)   [Variable]

This is the default display function used for idle mode. *BOX* is bounded about *WINDOW*, with bounces taking place every *WAIT* milliseconds. *BOX* can be a string, a bitmap, a window (whose image will be bounced about), or a list containing any number of these

(which will be cycled through). *BOX* defaults to the value of the variable IDLE.BOUNCING.BOX, which is initially a bitmap of the Venue logo. *WAIT* defaults to 1000 (one second).

Saving Virtual Memory State

Medley storage allocation occurs within a virtual memory space that is usually much larger than the physical memory on the computer. The virtual memory is stored as a large file on the computer's hard disk, called the virtual memory file. Medley controls the swapping of pages between this file and the real memory, swapping in virtual memory pages as they are accessed, and swapping out pages that have been modified. At any moment, the total state of the Medley virtual memory is stored partially in the virtual memory file, and partially in the real physical memory.

Medley provides facilities for saving the total state of the virtual memory, either on the virtual memory file, or in a file on an arbitrary file device. The function LOGOUT is used to write all altered (dirty) pages from the real memory to the virtual memory file and stop Medley, so that Medley can be restarted from the state of the LOGOUT. SAVEVM updates the virtual memory file without stopping Medley, which puts the virtual memory file into a consistant state (temporarily), so it could be restarted if the system crashes. SYSOUT and MAKESYS are used to save a copy of the total virtual memory state on a file, which can be loaded into another machine to restore Medley's state. VMEM.PURE.STATE can be used to "freeze" the current state of the virtual memory, so that Medley will come up in that state if it is restarted.

(**LOGOUT** *FAST*)   [Function]

> Stops Medley, and returns control to the operating system. If Medley is restarted, it should come up in the same state as when the LOGOUT was called. LOGOUT will not affect the state of open files.

> LOGOUT writes out all altered pages from real memory to the virtual memory file. If *FAST* is T, Medley is stopped without updating the virtual memory file. Note that after doing (LOGOUT T) it will not be possible to restart Medley from the point of the LOGOUT, and it may not be possible to restart it at all. Typing (LOGOUT T) is preferable to just booting the machine, because it also does other cleanup operations (closing network connections, etc.).

> If *FAST* is the symbol ?, LOGOUT acts like *FLG* = T if the virtual memory file is consistant, otherwise it acts like *FLG* = NIL. This insures that the virtual memory image can be restarted as of *some* previous state, not necessarily as of the LOGOUT.

(**SAVEVM** −)   [Function]

> This function is similar to logging out and continuing, but faster. It takes about as long as a logout, which can be as brief as 10 seconds or so if you have already written out most of your dirty pages by virtue of being idle a while. After the SAVEVM, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the SAVEVM) should there be a system crash or other disaster.

> If the system has been idle long enough (no keyboard or mouse activity), there are dirty pages to be written, and there are few enough dirty pages left to write that a SAVEVM would be quick, SAVEVM is automatically called. When SAVEVM is called automatically,

the cursor is changed to a special cursor: ⬚, stored in the variable SAVINGCURSOR. You can control how often SAVEVM is automatically called by setting the following two global variables:

SAVEVMWAIT   [Variable]
SAVEVMMAX   [Variable]

The system will call SAVEVM after being idle for SAVEVMWAIT seconds (initially 300) if there are fewer than SAVEVMMAX pages dirty (initially 600). These values are fairly conservative. If you want to be extremely wary, you can set SAVEVMWAIT = 0 and SAVEVMMAX = 10000, in which case SAVEVM will be called the first chance available after the first dirty page has been written.

The function SYSOUT saves the current state of Medley's virtual memory on a file, known as a "sysout file", or simply a "sysout". The file package can be used to save particular function definitions and other arbitrary objects on files, but SYSOUT saves the *total* state of the system. This capability can be useful in many situations: for creating customized systems for other people to use, or to save a particular system state for debugging purposes. Note that a sysout file can be very large (thousands of pages), and can take a long time to create, so it is not to be done lightly. The file produced by SYSOUT can be loaded into Medley's virtual memory and restarted to restore the virtual memory to the exact state that it had when the sysout file was made. The exact method of loading a sysout depend on the implementation. For more information on loading sysout files, see the users guide for your computer.

(**SYSOUT** *FILE*)   [Function]

Saves the current state of Medley's virtual memory on the file *FILE*, in a form that can be subsequently restarted. The current state of program execution is saved in the sysout file, so (PROGN (SYSOUT 'FOO) (PRINT 'HELLO)) will cause HELLO to be printed after the sysout file is restarted.

SYSOUT can take a very long time (ten or fifteen minutes), particularly when storing a file on a remote file server. To display some indication that something is happening, the cursor is changed to: ⬚. Also, as the sysout file is being written, the cursor is inverted line by line, to show that activity is taking place, and how much of the sysout has completed. For example, after the SYSOUT is about two-thirds done, the cursor would look like: ⬚. The SYSOUT cursor is stored in the variable SYSOUTCURSOR.

If *FILE* is non-NIL, the variable SYSOUTFILE is set to the body of *FILE*. If *FILE* is NIL, then the value of SYSOUTFILE instead. Therefore, (SYSOUT) will save the current state on the next higher version of a file with the same name as the previous SYSOUT. Also, if the extension for *FILE* is not specified, the value of SYSOUT.EXT is used. SYSOUT sets SYSOUTDATE (see the System Version Information section below) to (DATE), the time and date that the SYSOUT was performed.

If SYSOUT was not able to create the sysout file, because of disk or computer error, or because there was not enough space on the directory, SYSOUT returns NIL. Otherwise it returns the full file name of *FILE*.

Actually, SYSOUT "returns" twice; when the sysout file is first created, and when it is subsequently restarted. In the latter case, SYSOUT returns a list whose CAR is the full file name of *FILE*. For example, `(if (LISTP (SYSOUT 'FOO)) then (PRINT 'HELLO))` will cause HELLO to be printed when the sysout file is restarted, but not when SYSOUT is initially performed.

Note: SYSOUT does not save the state of any open files. Use WHENCLOSE (see the Closing and Reopening Files section in Chapter 24) to associate certain operations with open files so that when a SYSOUT is started up, these files will be reopened, and file pointers repositioned.

SYSOUT evaluates the expressions on BEFORESYSOUTFORMS (see also AROUNDEXITFNS) before creating the sysout file. This variable initially includes expressions to:

1. Set the variables SYSOUTDATE and SYSOUTFILE as described above

2. Default the sysout file name *FILE* according to the values of the variables SYSOUTFILE and SYSOUT.EXT, as described above

3. Perform any necessary operations on open files as specified by calls to WHENCLOSE.

After a sysout file is restarted (but *not* when it is initially created), SYSOUT evaluates the expressions on AFTERSYSOUTFORMS (see also AROUNDEXITFNS). This initially includes expressions to:

1. Perform any necessary operations on previously-opened files as specified by calls to WHENCLOSE

2. Possibly print a message, as determined by the value of SYSOUTGAG (see below)

3. Call SETINITIALS to reset the initials used for time-stamping (see the Time Stamps section of Chapter 16).

AROUNDEXITFNS   [Variable]

This variable provides a way to "advise" the system on cleanup and restoration activities to perform around LOGOUT, SYSOUT, MAKESYS and SAVEVM; It subsumes the functionality of BEFORESYSOUTFORMS, AFTERLOGOUTFORMS, etc. It's value is a list of functions (names) to call around every "exit" of the system. Each function is called with one argument, a symbol indicating which particular event is occuring. The symbols are:

BEFORLOGOUT The system is about to perform a LOGOUT.

BEFORESYSOUT
BEFOREMAKESYS
BEFORESAVEVM The system is about to perform a SYSOUT, MAKESYS or a SAVEVM.

AFTERLOGOUT
AFTERSYSOUT
AFTERMAKESYS
AFTERSAVEVM The system is starting up am image that was saved by performing a LOGOUT, SYSOUT, etc.

AFTERDOSYSOUT

AFTERDOMAKESYS
AFTERDOSAVEVM The system just made a copy of the virtual memory and saved it to disk. The image continues to run. These events only exist to allow you to negate the effects of saveing a copy of the virtual memory.

SYSOUTGAG   [Variable]

The value of SYSOUTGAG determines what is printed when a sysout file is restarted. If the value of SYSOUTGAG is a list, the list is evaluated, and no additional message is printed. This allows you to print a message. If SYSOUTGAG is non-NIL and not a list, no message is printed. Finally, if SYSOUTGAG is NIL (its initial value), and the sysout file is being restarted by the same user that made the sysout originally, you are greeted by printing the value of HERALDSTRING (see below) followed by a greeting message. If the sysout file was made by a different user, a message is printed, warning that the currently-loaded user init file may be incorrect for the current user (see the Greeting and Initialization Files section above).

(**MAKESYS** *FILE NAME*)   [Function]

Used to store a new Medley system on the "makesys file" *FILE*. Like SYSOUT, but before the file is made, the system is "initialized" by undoing the greet history, and clearing the display.

When the system is first started up, a "herald" is printed identifying the system, typically "Medley-XX DATE ...". If *NAME* is non-NIL, MAKESYS will use it instead of Medley-XX in the herald. MAKESYS sets HERALDSTRING to the herald string printed out.

MAKESYS also sets the variable MAKESYSDATE (see the next section below) to (DATE), i.e. the time and date the system was made.

Medley contains a routine that writes out dirty pages of the virtual memory during I/O wait, assuming that swapping has caused at least one dirty page to be written back into the virtual memory file (making it non-continuable). The frequency with which this routine runs is determined by:

BACKGROUNDPAGEFREQ   [Variable]

This variable determines how often the routine that writes out dirty pages is run. The *higher* BACKGROUNDPAGEFREQ is set, the *greater* the time between running the dirty page writing routine. Initially it is set to 4. The lower BACKGROUNDPAGEFREQ is set, the less responsiveness you get at typein, so it may not be desirable to set it all the way down to 1.

(**VMEM.PURE.STATE** *X*)   [NoSpread Function]

VMEM.PURE.STATE modifies the swapper's page replacement algorithm so that dirty pages are only written at the end of the virtual memory backing file. This "freezes" a given virtual memory state, so that Medley will come up in that state whenever it is restarted. This can be used to set up a "clean" environment on a pool machine, allowing each user to initialize the system simply by rebooting the computer.

The way to use VMEM.PURE.STATE is to set up the environment as you wish it to be "frozen," evaluate (VMEM.PURE.STATE T), and then call any function that saves the virtual memory state (LOGOUT, SAVEVM, SYSOUT, or MAKESYS). From that point on,

whenever the system is restarted, it will return to the state as of the saving operation. Future LOGOUT, SAVEVM, etc. operations will not reset this state.

Note: When the system is running in "pure state" mode, it uses a significant amount of the virtual memory backing file to save the "frozen" memory image, so this will reduce the amount of virtual memory space available for use.

(VMEM.PURE.STATE) returns T if the system is running in "pure state" mode, NIL otherwise.

(**REALMEMORYSIZE**)   [Function]

Returns the number of real memory pages in the computer.

(**VMEMSIZE**)   [Function]

Returns the number of pages in use in the virtual memory. This is the roughly the same as the number of pages required to make a sysout file on the local disk (see SYSOUT, above).

LASTVMEMFILEPAGE   [Variable]

Value is the total size of the virtual memory backing file. This variable is set when the system is started. You should *not* set it..

Note: When the virtual memory expands to the point where the virtual memory backing file is almost full, a break will occur with the warning message "Your virtual memory backing file is almost full. Save your work & reload asap." When this happens, it is strongly suggested that you save any important work and reload the system. If you continue working past this point, the system will start slowing down considerably, and it will eventually stop working.

System Version Information

Medley runs on a number of different machines, with many possible hardware configurations. There have been a number of different releases of the Medley software. These facts make it difficult to answer the important question "what software/hardware environment are you running?" when reporting bugs. The following functions allow the novice to collect this information.

(**PRINT-LISP-INFORMATION** *STREAM FILESTRING*)   [NoSpread Function]

Prints out a summary of the software and hardware environment that Medley is running in, and a list of all loaded patch files:

```
Venue Medley version
Medley 2.0 sysout of 7-Oct-92 15:18:52 on mips,
Emulator created: 20-Nov-92, memory size: 0,
machine d022899 mo
based on Envos Medley version Medley 2.0 sysout of 7-Oct-
92 15:18:52,
Make-init dates: 7-Oct-92 11:07:17, 7-Oct-92 11:26:22
Patch files: NIL
```

STREAM is the stream used to print the summary. If not given, it defaults to T.

FILESTRING is a string used to determine what loaded files should be listed as "patch files." All file names on LOADEDFILELST (see the Noticing Files section of Chapter 17) that have *FILESTRING* as a substring as listed. If *FILESTRING* is not given, it defaults to the string "PATCH".

(**CL:LISP-IMPLEMENTATION-TYPE**)   [Function]

Returns a string identifying the type of implementation that is running, e.g., "Medley".

(**CL:LISP-IMPLEMENTATION-VERSION**)   [Function]

Returns a string identifying the version that is running. Currently gives the system name and date, e.g., "KOTO of 10-Sep-85 08:25:46".

This uses the variables MAKESYSNAME and MAKESYSDATE (below), so it will change if you use MAKESYS (see the Saving Virtual Memory State section above) to create a custom sysout file, or explicitly changes these variables.

(**CL:SOFTWARE-TYPE**)   [Function]

Returns a string identifying the operating system that Interlisp is running under. Currently returns the string "Envos Medley".

(**CL:SOFTWARE-VERSION**)   [Function]

Returns a string identifying the version of the operating system that Interlisp is running under. Currently, this returns the date that the Medley release was originally created, so it doesn't change over MAKESYS or SYSOUT.

(**CL:MACHINE-TYPE**)   [Function]

Returns a string identifying the type of computer hardware that Medley is running on, i.e., "1108", "1132", "1186", "mips", etc.

(**CL:MACHINE-VERSION**)   [Function]

Returns a string identifying the version of the computer hardware that Medley is running on. Currently returns the microcode version and real memory size.

(**CL:MACHINE-INSTANCE**)   [Function]

Returns a string identifying the particular machine that Medley is running on. Currently returns the machine's NS address.

(**CL:SHORT-SITE-NAME**)   [Function]

Returns a short string identifying the site where the machine is located. Currently returns (ETHERHOSTNAME) (if non-NIL) or the string "unknown".

(**CL:LONG-SITE-NAME**)   [Function]

Returns a long string identifying the site where the machine is located. Currently returns the same as SHORT-SITE-NAME.

SYSOUTDATE [Variable]

> Value is set by SYSOUT (see the Saving Virtual Memory State section above) to the date before generating a virtual memory image file.

MAKESYSDATE [Variable]

> Value is set by MAKESYS (see the Saving Virtual Memory State section above) to the date before generating a virtual memory image file.

MAKESYSNAME [Variable]

> Value is a symbol identifying the release name of the current Medley system, e.g., :MEDLEY.

(**SYSTEMTYPE**) [Function]

> Allows programmers to write system-dependent code. SYSTEMTYPE returns a symbol corresponding to the implementation of Interlisp: D (for Medley), TOPS-20, TENEX, JERICO, or VAX.
>
> In Medley, (SELECTQ (SYSTEMTYPE) ...) expressions are expanded at compile time so that this is an effective way to perform conditional compilation.

(**MACHINETYPE**) [Function]

> Returns the type of machine that Medley is running on: either DORADO (for the Xerox 1132), DOLPHIN (for the Xerox 1100), DANDELION (for the Xerox 1108), DOVE (for the Xerox 1186), or MAIKO (for Unix, DOS, etc).

Date And Time Functions

(**DATE** *FORMAT*) [Function]

> Returns the current date and time as a string with format "*DD-MM-YY HH:MMM:SS*", where *DD* is day, *MM* is month, *YY* year, *HH* hours, *MMM* minutes, *SS* seconds, e.g., "7-Jun-85 15:49:34".
>
> If *FORMAT* is a date format as returned by DATEFORMAT (below), it is used to modify the format of the date string returned by DATE.

(**IDATE** *STR*) [Function]

> *STR* is a date and time string. IDATE returns *STR* converted to a number such that if DATE1 is before (earlier than) DATE2, then (IDATE *DATE1*) < (IDATE *DATE2*). If STR is NIL, the current date and time is used.
>
> Different Interlisp implementations can have different internal date formats. However, IDATE always has the essential property that (IDATE *X*) is less than (IDATE *Y*) if *X* is before *Y*, and (IDATE (GDATE *N*)) equals *N*. Programs which do arithmetic other than numerical comparisons between IDATE numbers may not work when moved from one implementation to another.

Generally, it is possible to increment an `IDATE` number by an integral number of days by computing a "1 day" constant, the difference between two convenient `IDATE`s, e.g. `(IDIFFERENCE (IDATE "2-JAN-80 12:00") (IDATE "1-JAN-80 12:00"))`. This "1 day" constant can be evaluated at compile time.

`IDATE` is guaranteed to accept as input the dates that `DATE` will output. It will ignore the parenthesized day of the week (if any). `IDATE` also correctly handles time zone specifications for those time zones registered in the list `TIME.ZONES` (below).

(**GDATE** *DATE FORMAT —*)   [Function]

Like `DATE`, except that *DATE* can be a number in internal date and time format as returned by `IDATE`. If *DATE* is `NIL`, the current time and date is used.

(**DATEFORMAT** *KEY₁ ... KEYₙ*)   [NLambda NoSpread Function]

`DATEFORMAT` returns a date format suitable as a parameter to `DATE` and `GDATE`. *KEY₁ ... KEYₙ* are a set of keywords (unevaluated). Each keyword affects the format of the date independently (except for `SLASHES` and `SPACES`). If the date returned by `(DATE)` with the default formatting was 7-Jun-85 15:49:34, the keywords would affect the formatting as follows:

NO.DATE	Doesn't include the date information, e.g. "`15:49:34`".
NUMBER.OF.MONTH	Shows the month as a number instead of a name, e.g. "`7-06-85 15:49:34`".
YEAR.LONG	Prints the year using four digits, e.g. "`7-Jun-1985 15:49:34`".
SLASHES	Separates the day, month, and year fields with slashes, e.g. "`7/Jun/85 15:49:34`".
SPACES	Separates the day, month, and year fields with spaces, e.g. "`7 Jun 85 15:49:34`".
NO.LEADING.SPACES	By default, the day field will always be two characters long. If `NO.LEADING.SPACES` is specified, the day field will be one character for dates earlier than the 10th, e.g. "`7-Jun-85 15:49:34`" instead of "`7-Jun-85 15:49:34`".
NO.TIME	Doesn't include the time information, e.g. "`7-Jun-85`".
TIME.ZONE	Includes the time zone in the time specification, e.g. "`7-Jun-85`".
NO.SECONDS	Doesn't include the seconds, e.g. "`7-Jun-85 15:49`".
DAY.OF.WEEK	Includes the day of the week in the time specification, e.g. "`7-Jun-85 15:49:34 PDT (Friday)`".

DAY.SHORT   If DAY.OF.WEEK is specified to include the day of the week, the week day is shortened to the first three letters, e.g. "7-Jun-85 15:49:34 PDT (Fri)". Note that DAY.SHORT has no effect unless DAY.OF.WEEK is also specified.

(**CLOCK** *N* —)   [Function]

If *N* = 0, CLOCK returns the current value of the time of day clock i.e., the number of milliseconds since last system start up.

If *N* = 1, returns the value of the time of day clock when you started up this Interlisp, i.e., difference between (CLOCK 0) and (CLOCK 1) is number of milliseconds (real time) since this Interlisp system was started.

If *N* = 2, returns the number of milliseconds of *compute* time since user started up this Interlisp (garbage collection time is subtracted off).

If *N* = 3, returns the number of milliseconds of compute time spent in garbage collections (all types).

(**SETTIME** *DT*)   [Function]

Sets the internal time-of-day clock. If *DT* = NIL, SETTIME attempts to get the time from the communications net; if it fails, you are prompted for the time. If *DT* is a string in a form that IDATE recognizes, it is used to set the time.

The following variables are used to interpret times in different time zones. \TimeZoneComp, \BeginDST, and \EndDST are normally set automatically if your machine is connected to a network with a time server. For standalone machines, it may be necessary to set them by hand (or in your init file, see the first section of this chapter) if you are not in the Pacific time zone.

TIME.ZONES   [Variable]

Value is an association list that associates time zone specifications (PDT, EST, GMT, etc.) with the number of hours west of Greenwich (negative if east). If the time zone specification is a single letter, it is appended to "DT" or "ST" depending on whether daylight saving time is in effect. Initially set to:

        ((8 . P) (7 . M) (6 . C) (5 . E) (0 . GMT))

This list is used by DATE and GDATE when generating a date with the TIME.ZONE format is specified, and by IDATE when parsing dates.

\TimeZoneComp   [Variable]

This variable should be initialized to the number of hours west of Greenwich (negative if east). For the U.S. west coast it is 8. For the east coast it is 5.

\BeginDST   [Variable]
\EndDST   [Variable]

\BeginDST is the day of the year on or before which Daylight Savings Time takes effect (i.e., the Sunday on or immediately preceding this day); \EndDST is the day on or before which Daylight Savings Time ends. Days are numbered with 1 being January 1, and

counting the days as for a leap year. In the USA where Daylight Savings Time is observed, \BeginDST = 121 and \EndDST = 305. In a region where Daylight Savings Time is not observed at all, set \BeginDST to 367.

Timers and Duration Functions

Often one needs to loop over some code, stopping when a certain interval of time has passed. Some systems provide an "alarm clock" facility, which provides an asynchronous interrupt when a time interval runs out. This is not particularly feasible in the current Medley environment, so the following facilities are supplied for efficiently testing for the expiration of a time interval in a loop context.

Three functions are provided: SETUPTIMER, SETUPTIMER.DATE, and TIMEREXPIRED?. There are also several new i.s.oprs: forDuration, during, untilDate, timerUnits, usingTimer, and resourceName (reasonable variations on upper/lower case are permissible).

These functions use an object called a timer, which encodes a future clock time at which a signal is desired. A timer is constructed by the functions SETUPTIMER and SETUPTIMER.DATE, and is created with a basic clock "unit" selected from among SECONDS, MILLISECONDS, or TICKS. The first two timer units provide a machine/system independent interface, and the latter provides access to the "real", basic strobe unit of the machine's clock on which the program is running. The default unit is MILLISECONDS.

Currently, the TICKS unit depends on what machine Medley is running on. The Xerox 1132 has about 1680 ticks per millisecond; the Xerox 1108 has about 34.746 ticks per millisecond; the Xerox 1185 and 1186 have about 62.5 ticks per millisecond. The advantage of using TICKS rather than one of the uniform interfaces is primarily speed; e.g., it may take over 400 microseconds to read the milliseconds clock (a software facility that uses the real clock), whereas reading the real clock itself may take less than ten microseconds. The disadvantage of the TICKS unit is its short roll-over interval (about 20 minutes) compared to the MILLISECONDS roll-over interval (about two weeks), and also the dependency on particular machine parameters.

(**SETUPTIMER** *INTERVAL OldTimer? timerUnits intervalUnits*) [Function]

SETUPTIMER returns a timer that will "go off" (as tested by TIMEREXPIRED?) after a specified time-interval measured from the current clock time. SETUPTIMER has one required and three optional arguments:

INTERVAL must be a integer specifying how long an interval is desired. *timerUnits* specifies the units of measure for the interval (defaults to MILLISECONDS).

If *OldTimer?* is a timer, it will be reused and returned, rather than allocating a new timer. *intervalUnits* specifies the units in which the *OldTimer?* is expressed (defaults to the value of *timerUnits*.

(**SETUPTIMER.DATE** *DTS OldTimer?*) [Function]

SETUPTIMER.DATE returns a timer (using the SECONDS time unit) that will "go off" at a specified date and time. *DTS* is a Date/Time string such as IDATE accepts (see the above section). If *OldTimer?* is a timer, it will be reused and returned, rather than allocating a new timer.

SETUPTIMER.DATE operates by first subtracting (IDATE) from (IDATE *DTS*), so there may be some large integer creation involved, even if *OLDTIMER?* is given.

(**TIMEREXPIRED?** *TIMER ClockValue.or.timerUnits*)   [Function]

If *TIMER* is a timer, and *ClockValue.or.timerUnits* is the time-unit of *TIMER*, TIMEREXPIRED? returns true if *TIMER* has "gone off".

ClockValue.or.timerUnits can also be a timer, in which case TIMEREXPIRED? compares the two timers (which must be in the same timer units). If X and Y are timers, then (TIMEREXPIRED? X Y) is true if X is set for an *earlier* time than Y.

There are a number of i.s.oprs that make it easier to use timers in iterative statements (see the Interative Statement section of Chapter 9). These i.s.oprs are given below in the "canonical" form, with the second "word" capitalized, but the all-caps and all-lower-case versions are also acceptable.

forDuration *INTERVAL*   [I.S. Operator]
during *INTERVAL*   [I.S. Operator]

INTERVAL is an integer specifying an interval of time during which the iterative statement will loop.

timerUnits *UNITS*   [I.S. Operator]

UNITS specifies the time units of the *INTERVAL* specified in forDuration.

untilDate *DTS*   [I.S. Operator]

DTS is a Date/Time string (such as IDATE accepts) specifying when the iterative statement should stop looping.

usingTimer *TIMER*   [I.S. Operator]

If usingTimer is given, *TIMER* is reused as the timer for forDuration or untilDate, rather than creating a new timer. This can reduce allocation if one of these i.s.oprs is used within another loop.

resourceName *RESOURCE*   [I.S. Operator]

RESOURCE specifies a resource name to be used as the timer storage (see the File Package Types section of Chapter 17). If *RESOURCE* = T, it will be converted to an internal name.

Some examples:

```
(during 6MONTHS timerUnits 'SECONDS
  until (TENANT-VACATED? HouseHolder)
  do (DISMISS <for-about-a-day>)
     (HARRASS HouseHolder)
  finally (if (NOT (TENANT-VACATED? HouseHolder))
             then (EVICT-TENANT HouseHolder)))
```

This example shows that how is is possible to have two termination condition: when the time interval of 6MONTHS has elapsed, or when the predicate (TENANT-VACATED? HouseHolder) becomes true. Note that the "finally" clause is executed regardless of which termination condition caused it.

Also note that since the millisecond clock will "roll over" about every two weeks, "6MONTHS" wouldn't be an appropriate interval if the timer units were the default case, namely MILLISECONDS.

```
(do (forDuration (CONSTANT (ITIMES 10 24 60 60 1000))
        do (CARRY.ON.AS.USUAL)
        finally (PROMPTPRINT "Have you had your 10-day check-up?")))
```

This infinite loop breaks out with a warning message every 10 days. One could question whether the millisecond clock, which is used by default, is appropriate for this loop, since it rolls-over about every two weeks.

```
(SETQ \RandomTimer (SETUPTIMER 0))
 (untilDate "31-DEC-83 23:59:59"  usingTimer \RandomTimer
   when (WINNING?) do (RETURN)
   finally (ERROR "You've been losing this whole year!"))
```

Here is a usage of an explicit date for the time interval; also, some stsorage has been squirreled away (as the value of \RandomTimer) for use by the call to SETUPTIMER in this loop.

```
(forDuration SOMEINTERVAL
   resourceName \INNERLOOPBOX
   timerunits 'TICKS
   do (CRITICAL.INNER.LOOP))
```

For this loop, you don't want any CONSing to take place, so \INNERLOOPBOX is defined as a resource which "caches" a timer cell (if it isn't already so defined), and wraps the entire statement in a WITH-RESOURCES call. Furthermore, a time unit of TICKS is specified, for lower overhead in this critical inner loop. In fact specifying a resourceName of T is the same as specifying it to be \ForDurationOfBox; this is just a simpler way to specify that a resource is wanted, without having to think up a name.

Resources

Medley is based on the use of a storage-management system which allocates memory space for new data objects, and automatically reclaims the space when no longer in use. More generally, Medley manages shared "resources", such as files, semaphors for processes, etc. The protocols for allocating and freeing such resources resemble those of ordinary storage management.

Sometimes you need to explicitly manage the allocation of resources. You may want the efficiency of explicit reclamation of certain temporary data; or it may be expensive to initialize a complex data object; or there may be an application that must not allocate new cells during some critical section of code.

The file manager type RESOURCES is available to help with the definition and usage of such classes of data; the definition of a RESOURCE specifies prototype code to do the basic management operations. The file manager command RESOURCES is used to save such definitions on files, and INITRESOURCES (see the Miscellaneous File Manager Commands section of Chapter 17) causes the initialization code to be output.

The basic needs of resource management are:

1. Obtaining a data item from the Lisp memory management system and configuring it to be a totally new instance of the resource in question

2. Freeing up an instance which is no longer needed

3. Getting an instance of the resource for temporary usage (whether "fresh" or a formerly freed-up instance)

4. Setting up any prerequisite global data structures and variables

A resources definition consists of four "methods": `INIT`, `NEW`, `GET`, and `FREE`; each "method" is a form that will specialize the definition for four corresponding user-level macros `INITRESOURCE`, `NEWRESOURCE`, `GETRESOURCE`, and `FREERESOURCE`. `PUTDEF` is used to make a resources definition, and the four components are specified in a proplist:

```
(PUTDEF
    'RESOURCENAME
    'RESOURCES
    '(NEW  NEW-INSTANCE-GENERATION-CODE
      FREE FREEING-UP-CODE
      GET  GET-INSTANCE-CODE
      INIT INITIALIZATION-CODE))
```

Each of the *xxx-CODE* forms is a form that will appear as if it were the body of a substitution macro definition for the corresponding macro (see the discussion on the macros below).

A Simple Example

Suppose one has several pieces of code which use a 256-character string as a scratch string. One could simply generate a new string each time, but that would be inefficient if done repeatedly. If you can guarantee that there are no re-entrant uses of the scratch string, then it could simply be stored in a global variable. However, if the code might be re-entrant on occasion, the program has to take precautions that two programs do not use the same scratch string at the same time. (This consideration becomes very important in a multi-process environment. It is hard to guarantee that two processes won't be running the same code at the same time, without using elaborate locks.) A typical tactic would be to store the scratch string in a global variable, and set the variable to `NIL` whenever the string is in use (so that re-entrant usages would know to get a "new" instance). For example, assuming the global variable `TEMPSTRINGBUFFER` is initialized to `NIL`:

```
[DEFINEQ (WITHSTRING NIL
    (PROG ((BUF (OR (PROG1 TEMPSTRINGBUFFER
                (SETQ TEMPSTRINGBUFFER NIL))
            (ALLOCSTRING 256))))
```

... use the scratch string in the variable `BUF` ...

```
        (SETQ TEMPSTRINGBUFFER BUF)
         (RETURN]
```

Here, the basic elements of a "resource" usage may be seen:

1. A call `(ALLOCSTRING 256)` allocates fresh instances of "buffer"

2. After usage is completed the instance is returned to the "free" state, by putting it back in the global variable `TEMPSTRINGBUFFER` where a subsequent call will find it

3. The prog-binding of `BUF` will get an existing instance of a string buffer if there is one -- otherwise it will get a new instance which will later be available for reuse

4. Some initialization is performed before usage of the resource (in this case, it is the setting of the global variable `TEMPSTRINGBUFFER`).

Given the following resources definition:

```
(PUTDEF
     'STRINGBUFFER
     'RESOURCES
     '(NEW   (ALLOCSTRING 256)
       FREE (SETQ TEMPSTRINGBUFFER (PROG1 . ARGS))
       GET  (OR (PROG1 TEMPSTRINGBUFFER
                     (SETQ TEMPSTRINGBUFFER NIL))
                (NEWRESOURCE TEMPSTRINGBUFFER)))
       INIT (SETQ TEMPSTRINGBUFFER NIL)))
```

we could then redo the example above as

```
(DEFINEQ (WITHSTRING NIL
     (PROG ((BUF (GETRESOURCE STRINGBUFFER)))
```

... use the string in the variable BUF ...

```
     (FREERESOURCE STRINGBUFFER BUF)
      (RETURN]
```

The advantage of doing the coding this way is that the resource management part of WITHSTRING is fully contained in the expansions of GETRESOURCE and FREERESOURCE, and thus there is no confusion between what is WITHSTRING code and what is resource management code. This particuar advantage will be multiplied if there are other functions which need a "temporary" string buffer; and of course, the resultant modularity makes it much easier to contemplate minor variations on, as well as multiple clients of, the STRINGBUFFER resource.

In fact, the scenario just shown above in the WITHSTRING example is so commonly useful that an abbreviation has been added; if a resources definition is made with *only* a NEW method, then appropriate FREE, GET, and INIT methods will be inferred, along with a coordinated globalvar, to be parallel to the above definition. So the above definition could be more simply written

```
(PUTDEF 'STRINGBUFFER
          'RESOURCES
          '(NEW   (ALLOCSTRING 256)))
```

and everything would work the same.

The macro WITH-RESOURCES simplifies the common scoping case, where at the beginning of some piece of code, there are one or more GETRESOURCE calls the results of which are each bound to a lambda variable; and at the ending of that code a FREERESOURCE call is done on each instance. Since the resources are locally bound to variables with the same name as the resource itself, the definition for WITHSTRING then simplifies to

```
(DEFINEQ (WITHSTRING NIL
     (WITH-RESOURCES (STRINGBUFFER)
```

 ... use the string in the variable STRINGBUFFER ...]

Trade-offs in More Complicated Cases

This simple example presumes that the various functions which use the resource are generally not re-entrant. While an occasional re-entrant use will be handled correctly (another example of the resource will simply be created), if this were to happen too often, then many of the resource requests will create and throw away new objects, which defeats one of the major purposes of using resources. A slightly more complex GET and FREE method can be of much benefit in maintaining a pool of available

resources; if the resource were defined to maintain a list of "free" instances, then the `GET` method could simply take one off the list and the `FREE` method could just push it back onto the list. In this simple example, the `SETQ` in the `FREE` method defined above would just become a "push", and the first clause of the `GET` method would just be (pop `TEMPSTRINGBUFFER`)

A word of caution: if the datatype of the resource is something very small that Medley is "good" at allocating and reclaiming, then explicit user storage management will probably not do much better than the combination of cons/createcell and the garbage collector. This would especially be so if more complicated `GET` and `FREE` methods were to be used, since their overhead would be closer to that of the built-in system facilities. Finally, it must be considered whether retaining multiple instances of the resource is a net gain; if the re-entrant case is truly rare, it may be more worthwhile to retain at most one instance, and simply let the instances created by the rarely-used case be reclaimed in the normal course of garbage collection.

Macros for Accessing Resources

Four user-level macros are defined for accessing resources:

(**NEWRESOURCE** *RESOURCENAME . ARGS*)	[Macro]
(**FREERESOURCE** *RESOURCENAME . ARGS*)	[Macro]
(**GETRESOURCE** *RESOURCENAME . ARGS*)	[Macro]
(**INITRESOURCE** *RESOURCENAME . ARGS*)	[Macro]

Each of these macros behave as if they were defined as a substitution macro of the form

$$((RESOURCENAME \ . \ ARGS) \ MACROBODY)$$

where the expression *MACROBODY* is selected by using the "code" supplied by the corresponding method from the *RESOURCENAME* definition.

Note that it is possible to pass "arguments" to your resource allocation macros. For example, if the `GET` method for the resource `FOO` is (`GETFOO . ARGS`), then (`GETRESOURCE FOO X Y`) is transformed into (`GETFOO X Y`). This form was used in the `FREE` method of the `STRINGBUFFER` resource described above, to pass the old `STRINGBUFFER` object to be freed.

(**WITH-RESOURCES** (*RESOURCE₁ RESOURCE₂ ...*) *FORM₁ FORM₂ ...*)	[Macro]

The `WITH-RESOURCES` macro binds lambda variables of the same name as the resources (for each of the resources *RESOURCE₁*, *RESOURCE₂*, etc.) to the result of the `GETRESOURCE` macro; then executes the forms *FORM₁*, *FORM₂*, etc., does a `FREERESOURCE` on each instance, and returns the value of the last form (evaluated and saved before the `FREERESOURCE`s).

Note: (`WITH-RESOURCES RESOURCE ...`) is interpreted the same as (`WITH-RESOURCES (RESOURCE) ...`). Also, the singular name `WITH-RESOURCE` is accepted as a synonym for `WITH-RESOURCES`.

Saving Resources in a File

Resources definitions may be saved on files using the `RESOURCES` file package command (see the Miscellaneous File Package Commands section of Chapter 17). Typically, one only needs the full definition available when compiling or interpreting the code, so it is appropriate to put the file package command in a (`DECLARE: EVAL@COMPILE DONTCOPY ...`) declaration, just as one might

do for a `RECORDS` declaration. But just as certain record declarations need *some* initialization in the run-time environment, so do most resources. This initialization is specified by the resource's `INIT` method, which is executed automatically when the resource is defined by the `PUTDEF` output by the `RESOURCES` command. However, if the `RESOURCES` command is in a `DONTCOPY` expression and thus is not included in the compiled file, then it is necessary to include a separate `INITRESOURCES` command (see the Miscellaneous File Manager Commands section of Chapter 17) in the filecoms to insure that the resource is properly initialized.

MEDLEY REFERENCE MANUAL

[This page intentionally left blank]

In most Common Lisp implementations, there is a "top-level read-eval-print loop," which reads an expression, evaluates it, and prints the results. In Medley, the Exec acts as the top-level loop, but does much more.

The Exec traps all THROWS, and recovers gracefully. It prints all values resulting from evaluation, on separate lines. (When zero values are returned, nothing is printed).

The Exec keeps track of your previous inputs, in the history list. Each entry you type creates a history event, which sotres the input and its values.

It's easy to use the results of earlier events, redo and event, or recall an earlier input, edit it, and run it. This makes it much easier to get your work done.

Multiple Execs and the Exec's Type

Sometimes you need more than one Exec open at a time. It's easy to open as many as you need by using the right button background menu and selecting the kind of Exec you need. The Execs are differentiated from one another by their "names" in their title bars and by their prompts. For example, the second Exec you open may have a prompt like 2/50> if it's the second Common Lisp Exec you've opened. Events in each Exec are placed on the global history list with their Exec number so the system can tell them apart.

Several variables are very important to an Exec since they control the format of reading and printing. Together these variables describe a type of exec, or its mode. Some standard bindings for the variables have been named to make mode setting easy. The names provide you with an Exec of the Common Lisp (LISP), Interlisp or Old Interlisp (IL), or Medley (XCL) type. An Exec's type is displayed in the title bar of its window:

```
Exec 2 (XCL)

2/53> *package*
#<Package XCL-USER>
2/56> *readtable*
#<ReadTable XCL/74,151670>
2/57>
```

A Brief Example of Exec Interactions

The following dialogue contains examples and gives the flavor of the use of an Exec. The commands are described in greater detail in the following sections. For now, be sure to type these examples to an Exec whose *PACKAGE* is set to the XCL-USER package. The Exec that Medley starts up with is set to the XCL-USER package. Each prompt consists of an Exec number, an event number and a prompt character (">" for Common Lisp and "←" for Interlisp).

```
Exec 2 (LISP)

2/1188> (SETQ FOO 5)
5
2/1189> (SETQ FOO 10)
10
2/1190> UNDO SETQ
SETQ undone.
2/1191> FOO
5
2/1192>
```

You have instructed the Exec to UNDO the previous event.

```
Exec 2 (LISP)

2/1192> SET(LST1 (A B C))
(A B C)
2/1195> (SETQ LST2 '(D E F ))
(D E F)
2/1196> (MAPC #'(LAMBDA (X) (SETF (GET
X 'MYPROP) T)) LST1)
(A B C)
2/1197>
```

The Exec accepts input both in APPLY format (the SET) and EVAL format (the SETQ). In event 1196, you added a property MYPROP to the symbols A, B, and C.

```
Exec 2 (LISP)

2/1192> SET(LST1 (A B C))
(A B C)
2/1195> (SETQ LST2 '(D E F ))
(D E F)
2/1196> (MAPC #'(LAMBDA (X) (SETF (GET
X 'MYPROP) T)) LST1)
(A B C)
2/1197> USE LST2 FOR LST1 IN 1196
(D E F)
2/1198>
```

You told the Exec to go back to event 1196, substitute LST2 for LST1, and then re-execute the expression.

```
Exec 2 (XCL)
NIL
2/48> (setf myhash (make-hash-table))
#<Hash-Table @ 361,117340>
2/49> (setf (gethash 'foo myhash)(string
'foo))
"FOO"
2/50>
```

If STRING were computationally expensive (it isn't), you might be caching its value for later use.

```
Exec 2 (XCL)
2/48> (setf myhash (make-hash-table))
#<Hash-Table @ 361,117340>
2/49> (setf (gethash 'foo myhash)(string
'foo))
"FOO"
2/50> use fie for foo in string
"FIE"
2/51>
```

You now decide you would like to redo the SETF with a different value. You can specify the event using any symbol in the expression.

```
Exec 2 (XCL)

2/68> ?? setf

        USE FIE FOR FOO IN STRING
2/50>   (SETF (GETHASH (QUOTE FIE) MYH
ASH) (STRING (QUOTE FIE)))
        "FIE"


2/69>▲
```

Here you ask the Exec (using the `??` command) what it has on its history list for the last input. Since the event corresponds to a command, the Exec displays both the original command and the generated input.

You'll usually deal with the Exec at top level or in the debugger, where you type in expressions for evaluation, and see the values printed out. An Exec acts much like a standard Lisp top-level loop, but before it evaluates an input, it first adds it to the history list. If the operation is aborted or causes an error, the input is still available for you to modify or re-execute.

After updating the history list, the Exec executes the computation (i.e., evaluates the form or applies the function to its arguments), saves the value in the history-list entry for that input, and prints the result. Finally the Exec displays a prompt to show it's again ready for input.

Input Formats

The Exec accepts three forms of input: an expression to be evaluated (`EVAL`-format), a function-name and arguments to apply it to (`APPLY`-format), and Exec commands, as follows:

`EVAL`-format input   If you type a single expression, either followed by a carriage-return, or, in the case of a list, terminated with balanced parenthesis, the expression is evaluated and the value is returned. For example, if the value of `FOO` is the list `(A B C)`:

```
Exec 3 (XCL)

3/133> foo
(A B C)
3/134>
```

Similarly, if you type a Lisp expression, beginning with a left parenthesis and terminated by a matching right parenthesis, the form is simply passed to `EVAL` for evaluation. Notice that it is not necessary to type a carriage return at the end of such a form; the reader will supply one automatically. If a carriage-return is typed before the final matching right parenthesis or bracket, it is treated the same as a space, and input continues. The following examples are interpreted identically:

```
Exec 3 (XCL)

3/38> (+ 1 (* 2 3))
7
3/40> (+ 1 (*
2 3))
7
3/41>
```

`APPLY`-format input   Often, you call functions with constant argument values, which would have to be quoted if you typed them in `EVAL`-format. For convenience, if you type a symbol immediately followed by a list, the symbol is `APPLY`ed to the elements within the list, unevaluated. The input is terminated by the matching right parenthesis. For example, typing `LOAD(FOO)` is equivalent to typing `(LOAD 'FOO)`, and `GET(X`

`COLOR)` is equivalent to `(GET 'X 'COLOR)`. As a simple special case, a single right parenthesis is treated as a balanced set of parentheses, e.g. `UNBREAK)` is equivalent to `UNBREAK()`

The reader will only supply the "carriage return" automatically if no space appears between the initial symbol and the list that follows; if there is a space after the initial symbol on the line and the list that follows, the input is not terminated until you type a carriage return.

The Exec will not consider unparenthesized input with more than one argument to be in apply format, e.g.:

`LIST(1)` is apply format (executes after closing parenthesis is typed)

`LIST (1)` is apply format (second argument is a list, no trailing arguments given)

`LIST '(1) 2 3` is NOT apply format, arguments are evaluated

`LIST 1 2 3` is NOT apply format, arguments are evaluated

`LIST 1`not legal input: second argument is not a list

Note that `APPLY`-format input cannot be used for macros or special forms.

Exec commands   The Exec recognizes a number of commands, which usually refer to past events on the history list. These commands are treated specially; for example, they may not be put on the history list. The format of a command is always a line beginning with the command name. (The Exec looks up the command name independent of package.) The remainder of the line, if any, is treated as "arguments" to the command. For example,

```
128> UNDO
mapc undone
129> UNDO (FOO --)
foo undone
```

are both valid command inputs.

Event Specification

Exec commands, like `UNDO`, frequently refer to previous events in the session's history. All Exec commands use the same conventions and syntax for indicating which event(s) the command refers to. This section shows you the syntax used to specify previous events.

An event address identifies one event on the history list. For example, the event address 42 refers to the event with event number 42, and -2 refers to two events back in the current Exec. Usually, an event address will contain only one or two commands.

Event addresses can be concatenated. For example, if `FOO` refers to event N, `FOO FIE` will refer to the first event before event N which contains `FIE`.

The symbols used in event addresses (such as `AND`, `F`, etc.) are compared with `STRING-EQUAL`, so that it does not matter what the current package is when you type an event address symbol to an Exec.

Specifications used below of the form *EventAddress* refer to event addresses, as described above. Since an event address may contain multiple words, the event address is parsed by searching for the words which delimit it. For example, in *EventAddress₁* AND *EventAddress₂*, the notation *EventAddress₁* corresponds to all words up to the AND in the event specification, and *EventAddress₂* to all words after the AND in the event specification.

Event addresses are interpreted as follows:

N (an integer)   If N is positive, it refers to the event with event number N (no matter which Exec the event occurred in.) If N is negative, it always refers to the event -N events backwards, counting only events belonging to the current Exec.

F   Specifies that the next object in the event address is to be searched for, regardless of what it is. For example, F -2 looks for an event containing -2.

FROM *EventAddress*

All events since *EventAddress*, inclusive. For example, if there is a single Exec and the current event is number 53, then FROM 49 specifies events 49, 50, 51, and 52. FROM includes events from *all* Execs.

ALL *EventAddress*

Specifies all events satisfying *EventAddress*. For example, ALL LOAD, ALL SUCHTHAT FOO-P.

empty   If nothing is specified, it is the same as specifying -1, i.e., the last event in the current Exec.

EventSpec₁ AND *EventSpec₂* AND . . . AND *EventSpecₙ*

Each of the is an event specification. The lists of events are concatenated. For example, REDO ALL MAPC AND ALL STRING AND 32 redoes all events containing MAPC, all containing STRING, and also event 32. Duplicate events are removed.

Exec Commands

You enter an Exec commands by typing the name of the command at the prompt. The name of an Exec command is not a symbol and therefore is not sensitive to the setting of the current package (the value of *PACKAGE*).

EventSpec is used to denote an event specification which in most cases will be either a specific event address (e.g., 42) or a relative one (e.g., -3). Unless specified otherwise, omitting *EventSpec* is the same as specifying *EventSpec* = -1. For example, REDO and REDO -1 are the same.

REDO *EventSpec*   [Exec command]

Redoes the event or events specified by *EventSpec*. For example, REDO 123 redoes the event numbered 123.

RETRY *EventSpec*   [Exec command]

> Like REDO but sets the debugger parameters so that any errors that occur while executing *EventSpec* will cause breaks.

USE *NEW* [**FOR** *OLD*] [**IN** *EventSpec*]   [Exec command]

> Substitutes *NEW* for *OLD* in the events specified by *EventSpec*, and redoes the result. *NEW* and *OLD* can include lists or symbols, etc.
>
> For example, USE SIN (- X) FOR COS X IN -2 AND -1 will substitute SIN for every occurrence of COS in the previous two events, and substitute (- X) for every occurrence of X, and reexecute them. (The substitutions do not change the previous information saved about these events on the history list.)
>
> If IN *EventSpec* is omitted, the first member of *OLD* is used to search for the appropriate event. For example, USE DEFAULTFONT FOR DEFLATFONT is equivalent to USE DEFAULTFONT FOR DEFLATFONT IN F DEFLATFONT. The F is inserted to handle the case where the first member of *OLD* could be interpreted as an event address command.
>
> If *OLD* is omitted, substitution is for the "operator" in that command. For example FBOUNDP(FF) followed by USE CALLS is equivalent to USE CALLS FOR FBOUNDP IN -1.
>
> If *OLD* is not found, USE will print a question mark, several spaces and the pattern that was not found. For example, if you specified USE Y FOR X IN 104 and X was not found, "X   ?" is printed to the Exec.
>
> You can also specify more than one substitution simultaneously as follows:

USE *NEW₁* **FOR** *OLD₁* **AND** ... **AND** *NEWₙ* **FOR** *OLDₙ* [**IN** *EventSpec*]   [Exec command]

> [The USE command is parsed by a small finite state parser to distinguish the expressions and arguments. For example, USE FOR FOR AND AND AND FOR FOR will be parsed correctly.]
>
> Every USE command involves three pieces of information: the expressions to be substituted, the arguments to be substituted for, and an event specification that defines the input expression in which the substitution takes place. If the USE command has the same number of expressions as arguments, the substitution procedure is straightforward. For example, USE X Y FOR U V means substitute X for U and Y for V, and is equivalent to USE X FOR U AND Y FOR V.
>
> However, the USE command also permits distributive substitutions for substituting several expressions for the same argument. For example, USE A B C FOR X means first substitute A for X then substitute B for X (in a new copy of the expression), then substitute C for X. The effect is the same as three separate USE commands.
>
> Similarly, USE A B C FOR D AND X Y Z FOR W is equivalent to USE A FOR D AND X FOR W, followed by USE B FOR D AND Y FOR W, followed by USE C FOR D AND Z FOR W. USE A B C FOR D AND X FOR Y also corresponds to three substitutions, the first with A for D and X for Y, the second with B for D, and X for Y, and the third with C

for `D`, and again `X` for `Y`. However, `USE A B C FOR D AND X Y FOR Z` is ambiguous and will cause an error.

Essentially, the `USE` command operates by proceeding from left to right handling each `AND` separately. Whenever the number of expressions exceeds the available expressions, multiple `USE` expressions are generated. Thus `USE A B C D FOR E F` means substitute `A` for `E` at the same time substituting `B` for `F`, then in another copy of the indicated expression, substitute `C` for `E` and `D` for `F`. This is also equivalent to `USE A C FOR E AND B D FOR F`.

The `USE` command correctly handles the situation where one of the old expressions is the same as one of the new ones, `USE X Y FOR Y X`, or `USE X FOR Y AND Y FOR X`.

? *NAME* [Exec command]

If *NAME* is not provided describes all available Exec commands by printing the name, argument list, and description of each. With *NAME*, only that command is described.

?? *EventSpec* [Exec command]

Prints the most recent event matching the given *EventSpec*. Without *EventSpec*, lists all entries on the history list from all execs, not necessarily in the order in which they occured (since the list is in allocation order). If you haven't completed typing a command it will be listed as "<in progress>" .

Note: Event nubmers are allocated at the time the prompt is printed, except in the Old Interlisp exec where they are assigned at the end of type-in. This means that if activity occurs in another exec, the number printed next to the command is not necessarily the number associated with the event.

CONN *DIRECTORY* [Exec command]

Changes default pathname to *DIRECTORY*.

DA [Exec command]

Returns current date and time.

DIR *PATHNAME KEYWORDS* [Exec command]

Shows a directory listing for *PATHNAME* or the connected directory. If provided, *KEYWORDS* indicate information to be displayed for each file. Some keywords are: *AUTHOR*, *AU*, *CREATIONDATE*, *DA*, etc.

DO-EVENTS *INPUTS ENV* [Exec command]

DO-EVENTS is intended as a way of putting together several different events, which can include commands. It executes the multiple *INPUTS* as a single event. The values returned by the *DO-EVENTS* event are the concatenation of the values of the inputs. An input is not an EventSpec, but a call to a function or command. If *ENV* is provided it is a lexical environment in which all evaluations (functions and commands) will take place. Event specification in the *INPUTS* should be explicit, not relative, since referring to the last event will reinvoke the executing *DO-EVENTS* command.

FIX *EventSpec* [Exec command]

Edits the specified event prior to re-executing it. If the number of characters in the fixed line is less than the variable TTYINFIXLIMIT then it will be edited using TTYIN, otherwise the Lisp editor is called via EDITE.

FORGET *EventSpec* [Exec command]

Erases UNDO information for the specified events.

NAME *COMMAND-NAME ARGUMENTS EVENT-SPEC* [Exec command]

Defines a new command, *COMMAND-NAME*, and its *ARGUMENTS*, containing the events in *EVENT-SPEC*.

NDIR *PATHNAME KEYWORDS* [Exec command]

Shows a directory listing for *PATHNAME* or the connected directory in abbreviated format. If provided, *KEYWORDS* indicate information to be displayed for each file. Some keywords are: AUTHOR, AU, CREATIONDATE, DA, etc.

PL *SYMBOL* [Exec command]

Prints the property list of *SYMBOL* in an easy to read format.

REMEMBER *&REST EVENT-SPEC* [Exec command]

Tells File Manager to remember type-in from specified event(s), *EVENT-SPEC*, as expressions to save.

SHH *LINE* [Exec command]

Executes *LINE* without history list processing.

UNDO *EventSpec* [Exec command]

Undoes the side effects of the specified event (see below under "Undoing").

PP *NAME TYPES* [Exec command]

Shows (prettyprinted) the definitions for *NAME* specified by *TYPES*.

SEE *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, hiding comments.

SEE* *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, showing comments.

TIME *FORM &KEY REPEAT &ENVIRONMENT ENV* [Exec command]

Times the evaluation of *FORM* in the lexical environment *ENV*, repeating *REPEAT* number of times. Information is displayed in the Exec window.

TY *FILES* [Exec command]

Exactly like the *TYPE* Exec command.

TYPE *FILES*   [Exec command]

> Prints the contents of *FILES* in the Exec window, hiding comments.

Variables

A number of variables are provided for convenience in the Exec.

IL:IT   [Variable]

> Whenever an event is completed, the global value of the variable IT is reset to the event's value. For example,


```
Exec 3 (XCL)

3/41> (sqrt 2)
1.4142135
3/43> (sqrt il:it)
1.1892071
3/44>
```

> Following a ?? command, IL:IT is set to the value of the last event printed. The inspector has an option for setting the variable IL:IT to the current selection or inspected object, as well. The variable IL:IT is global, and is shared among all Execs. IL:IT is a convenient mechanism for passing values from one process to another.
>
> **Note**: IT is in the Interlisp package and these examples are intended for an Exec whose *PACKAGE* is set to XCL-USER. Thus, IT must be package qualified (the IL:).

The following variables are maintained independently by each Exec. (When a new Exec is started, the initial values are NIL, or, for a nested Exec, the value for the "parent" Exec. However, events executed under a nested Exec will not affect the parent values.)

CL:-   [Variable]
CL:+   [Variable]
CL:++   [Variable]
CL:+++   [Variable]

> While a form is being evaluated by the Exec, the variable CL:- is bound to the form, CL:+ is bound to the previous form, CL:++ the one before, etc. If the input is in apply-format rather than eval-format, the value of the respective variable is just the function name.

CL:*   [Variable]
CL:**   [Variable]
CL:***   [Variable]

> While a form is being evaluated by the Exec, the variable CL:* is bound to the (first) value returned by the last event, CL:** to the event before that, etc. The variable CL:* differs from IT in that IT is global while each separate Exec maintains its own copy of CL:*, CL:** and CL:***. In addition, the history commands change IT, but only inputs that are retained on the history list can change CL:*.

CL:/   [Variable]
CL://   [Variable]
CL:///   [Variable]

> While a form is being evaluated by an Exec, the variable `CL:/` is bound to a list of the results of the last event in that Exec, `CL://` to the values of the event before that, etc.

Fonts in the Exec

The Exec can use different fonts for displaying the prompt, user's input, intermediate printout, and the values returned by evaluation. The following variables control the Exec's font use:

> **PROMPTFONT**   [Variable]
>
> > Font used for printing the event prompt.
>
> **INPUTFONT**   [Variable]
>
> > Font used for echoing your type-in.
>
> **PRINTOUTFONT**   [Variable]
>
> > Font used for any intermediate printing caused by execution of a command or evaluation of a form. Initially the same as `DEFAULTFONT`.
>
> **VALUEFONT**   [Variable]
>
> > Font used to print the values returned by evaluation of a form. Initially the same as `DEFAULTFONT`.

Modifying an Exec

> (**CHANGESLICE** *N HISTORY —*)   [Function]
>
> > Changes the maximum number of events saved on the history list *HISTORY* to *N*. If `NIL`, *HISTORY* defaults to the top level history `LISPXHISTORY`.
> >
> > The effect of *increasing* the time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* the time-slice will immediately remove a sufficient number of the older events to bring the history list down to the proper size. However, `CHANGESLICE` is undoable, so that these events are (temporarily) recoverable. Therefore, if you want to recover the storage associated with these events without waiting *N* more events until the `CHANGESLICE` event drops off the history list, you must perform a `FORGET` command.

Defining New Commands

You can define new Exec commands using the `XCL:DEFCOMMAND` macro.

> (**XCL:DEFCOMMAND** *NAME ARGUMENT-LIST &REST BODY*)   [Macro]
>
> > `XCL:DEFCOMMAND` is like `XCL:DEFMACRO`, but defines new Exec commands. The *ARGUMENT-LIST* can have keywords, and use all of the features of macro argument lists. When *NAME* is subsequently typed to the Exec, the rest of the line is processed like the arguments to a macro, and the *BODY* is executed. `XCL:DEFCOMMAND` is a definer; the

File Manager will remember typed-in definitions and allow them to be saved, edited with `EDITDEF`, etc.

There are three kinds of commands that can be defined, `:EVAL`, `:QUIET`, and `:INPUT`. Commands can also be marked as only for the debugger, in which case they are labelled as `:DEBUGGER`. The command type is noted by supplying a list for the *NAME* argument to `XCL:DEFCOMMAND`, where the first element of the list is the command name, and the other elements are keyword(s) for the command type and, optionally `:DEBUGGER`.

The documentation string in user defined Exec commands is automatically added to the documentation descriptions by the `CL:DOCUMENTATION` function under the `COMMANDS` type and can be shown using the `?` Exec command.

`:EVAL`   This is the default. The body of the command just gets executed, and its value is the value of the event. For example (in an XCL Exec),

```
Exec 3 (XCL)

3/47> (DEFCOMMAND (LS :EVAL)
        (&OPTIONAL (NAMESTRING
*DEFAULT-PATHNAME-DEFAULTS*)
        &REST DIRECTORY-KEYWORDS)
        (MAPC #'(LAMBDA (PATHNAME)
          (FORMAT T "~&~A" (NAMESTRING PATHNAME)))
        (APPLY #'DIRECTORY NAMESTRING
DIRECTORY-KEYWORDS))
  (VALUES))
LS
3/48>
```

would define the `LS` command to print out all file names that match the input `NAMESTRING`. The `(VALUES)` means that no value will be printed by the event, only the intermediate output from the `FORMAT`.

`:QUIET`   These commands are evaluated, but neither your input nor the results of the command are stored on the history list. For example, the `??` and `SHH` commands are quiet.

`:INPUT`   These commands work more like macros, in that the result of evaluating the command is treated as a new line of input. The `FIX` command is an input command. The result is treated as a line; a single expression in `EVAL`-format should be returned as a list of the expression to `EVAL`.

Undoing

Note: This discussion only applies to undoing under the Exec or Debugger, and within the `UNDOABLY` macro; text and structure editors handle undoing differently.

The `UNDO` facility allows recording of destructive changes such that they can be played back to restore a previous state. There are two kinds of `UNDO`ing: one is done by the Exec, the other is available for use in your code. Both methods share information about what kind of operations can be undone and where the changes are recorded.

Undoing in the Exec

UNDO *EventSpec* [Exec command]

The Exec's UNDO command is implemented by watching the evaluation of forms and requiring undoable operations in that evaluation to save enough information on the history list to reverse their side effects. The Exec simply executes operations, and any undoable changes that occur are automatically saved on the history list by the responsible functions. The UNDO command works on itself the same way: it recovers the saved information and performs the corresponding inverses. Thus, UNDO is effective on itself, so that you can UNDO an UNDO, and UNDO that, etc.

Only when you attempt to undo an operation does the Exec check to see whether any information has been saved. If none has been saved, and you have specifically named the event you want undone, the Exec types nothing saved. (When you just type UNDO, the Exec only tries to undo the last operation.)

UNDO watches evaluation using CL:EVALHOOK (thus, calling CL:EVALHOOK cannot be undone). Each form given to EVAL is examined against the list LISPXFNS to see if it has a corresponding undoable version. If an undoable version of a call is found, it is called with the same arguments instead of the original. Therefore, before evaluating all subforms of your input, the Exec substitutes the corresponding undoable call for any destructive operation. For example, if you type (DEFUN FOO ...), undoable versions of the forms that set the definition into the symbol function cell are evaluated. FOO's function definition itself is not made undoable.

Undoing in Programs

There are two ways to make a program undoable. The simplest method is to wrap the program's form in the UNDOABLY macro. The other is to call undoable versions of destructive operations directly.

(**XCL:UNDOABLY** *&REST FORMS*) [Macro]

Executes the forms in *FORMS* using undoable versions of all destructive operations. This is done by "walking" (see WALKFORM) all of the *FORMS* and rewriting them to use the undoable versions of destructive operations (LISPXFNS makes the association).

(**STOP-UNDOABLY** *&REST FORMS*) [Macro]

Normally executes as PROGN; however, within an UNDOABLY form, explicitly causes *FORMS* not to be done undoably. Turns off rewriting of the *FORMS* to be undoable inside an UNDOABLY macro.

Undoable Versions of Common Functions

When efficiencyis a serious concern, you may need more control over the saving of undo information than that provided by the UNDOABLY macro.

To make a function undoable, you can simply substitute the corresponding undoable function in your program. When the undoable function is called, it will save the undo information in the current event on the history list.

Various operations, most notably `SETF`, have undoable versions. The following undoable macros are initially available:

```
UNDOABLY-POP                    UNDOABLY-SET-SYMBOL
UNDOABLY-PUSH                   UNDOABLY-MAKUNBOUND
UNDOABLY-PUSHNEW                UNDOABLY-FMAKUNBOUND
UNDOABLY-REMF                   UNDOABLY-SETQ
UNDOABLY-ROTATEF                XCL:UNDOABLY-SETF
UNDOABLY-SHIFTF                 UNDOABLY-PSETF
UNDOABLY-DECF                   UNDOABLY-SETF-SYMBOL-FUNCTION
UNDOABLY-INCF                   UNDOABLY-SETF-MACRO-FUNCTION
```

Note: Many destructive Common Lisp functions do not have undoable versions, e.g., `CL:NREVERSE`, `CL:SORT`, etc. You can see the current list of undoable functions on the association list `LISPXFNS`.

Modifying the UNDO Facility

You may want to extend the `UNDO` facility after creating a form whose side effects might be undoable, for instance a file renaming function.

You need to write an undoable version of the function. You can do this by explicitly saving previous state information, or by renaming calls in the function to their undoable equivalent. Undo information should be saved on the history list using `IL:UNDOSAVE`.

You must then hook the undoable version of the function into the undo facility. You do this by either using the `IL:LISPXFNS` association list, or in the case of a `SETF` modifier, on the `IL:UNDOABLE-SETF-INVERSE` property of the `SETF` function.

`LISPXFNS`   [Variable]

> Contains an association list that maps from destructive operations to their undoable form. Initially this list contains:
>
> ```
> ((CL:POP . UNDOABLY-POP)
>  (CL:PSETF . NDOABLY-PSETF)
>  (CL:PUSH . UNDOABLY-PUSH)
>  (CL:PUSHNEW . UNDOABLY-PUSHNEW)
>  ((CL:REMF) . UNDOABLY-REMF)
>  (CL:ROTATEF . UNDOABLY-ROTATEF)
>  (CL:SHIFTF . UNDOABLY-SHIFTF)
>  (CL:DECF . UNDOABLY-DECF)
>  (CL:INCF . UNDOABLY-INCF)
>  (CL:SET . UNDOABLY-SET-SYMBOL)
>  (CL:MAKUNBOUND . UNDOABLY-MAKUNBOUND)
>  (CL:FMAKUNBOUND . UNDOABLY-FMAKUNBOUND)
>  . . . plus the original Interlisp undo associations)
> ```

(**`XCL:UNDOABLY-SETF`** *PLACE VALUE ...*)   [Macro]

> Like `CL:SETF` but saves information so it may be undone.   `UNDOABLY-SETF` uses undoable versions of the `SETF` function located on the `UNDOABLE-SETF-INVERSE` property of the function being `SETF`ed. Initially these `SETF` names have such a property:
>
> ```
> CL:SYMBOL-FUNCTION - UNDOABLY-SETF-SYMBOL-FUNCTION
> CL:MACRO-FUNCTION - UNDOABLY-SETF-MACRO-FUNCTION
> ```

(**UNDOABLY-SETQ** *&REST FORMS*)   [Function]

> Typed-in SETQs (and SETFs on symbols) are made undoable by substituting a call to
> UNDOABLY-SETQ.   UNDOABLY-SETQ operates like SETQ on lexical variables or those
> with dynamic bindings; it only saves information on the history list for changes to global,
> "top-level" values.

(**UNDOSAVE** *UNDOFORM HISTENTRY*)   [Function]

> Adds the undo information *UNDOFORM* to the SIDE property of the history event
> *HISTENTRY*. If there is no SIDE property, one is created. If the value of the SIDE
> property is NOSAVE, the information is not saved.   *HISTENTRY* specifies an event. If
> *HISTENTRY*=NIL, the value of LISPXHIST is used. If both *HISTENTRY* and LISPXHIST
> are NIL, UNDOSAVE is a no-op.

> The form of UNDOFORM is (FN . ARGS). Undoing is done by performing (APPLY (CAR
> UNDOFORM) (CDR UNDOFORM)).

\#UNDOSAVES   [Variable]

> The maximum number of UNDOFORMs to be saved for a single event. When the count of
> UNDOFORMs reaches this number, UNDOSAVE prints the message CONTINUE SAVING?,
> asking if you want to continue saving. If you answer NO or default, UNDOSAVE discards
> the previously saved information for this event, and makes NOSAVE be the value of the
> property SIDE, which disables any further saving for this event. If you answer YES,
> UNDOSAVE changes the count to -1, which is then never incremented, and continues
> saving. The purpose of this feature is to avoid tying up large quantities of storage for
> operations that will never need to be undone.

> If \#UNDOSAVES is negative, then when the count reaches (ABS \#UNDOSAVES),
> UNDOSAVE simply stops saving without printing any messages or other interactions.
> \#UNDOSAVES = NIL is equivalent to \#UNDOSAVES = infinity. \#UNDOSAVES is initially
> NIL.

> The configuration described here is very satisfactory. You pay a very small price for the
> ability to undo what you type in, since the interpreted evaluation is simply watched for
> destructive operations, or if you wish to protect yourself from malfunctioning in your
> own programs, you can explicitly call, or rewrite your program to explicitly call, undoable
> functions.

Undoing Out of Order

UNDOABLY-SETF operates undoably by saving (on the history list) the cell that is to be changed and
its original contents. Undoing an UNDOABLY-SETF restores the saved contents.

This implementation can produce unexpected results when multiple modifications are made to the
same piece of storage and then undone out of order. For example, if you type (SETF (CAR FOO)
1), followed by (SETF (CAR FOO) 2), then undo both events by undoing the most recent event
first, then undoing the older event, FOO will be restored to its state before either event operated.
However if you undo the first event, then the second event, (CAR FOO) will be 1, since this is what
was in CAR of FOO before (UNDOABLY-SETF (CAR FOO) 2) was executed. Similarly, if you type

(NCONC FOO '(1)), followed by (NCONC FOO '(2)), undoing just (NCONC FOO '(1)) will remove both 1 and 2 from FOO. The problem in both cases is that the two operations are not independent.

In general, operations are always independent if they affect different lists or different sublists of the same list. Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the right thing. However, undoing dependent operations out of order may not always have the predicted effect.

Format and Use of the History List

LISPXHISTORY [Variable]

The Exec currently uses one primary history list, LISPXHISTORY for the storing events.

The history list is in the form (*EVENTS EVENT# SIZE MOD*), where *EVENTS* is a list of events with the most recent event first, *EVENT#* is the event number for the most recent event on *EVENTS*, *SIZE* is the the maximum length *EVENTS* is allowed to grow. *MOD* is is the maximum event number to use, after which event numbers roll over. *LISPXHISTORY* is initialized to (NIL 0 100 1000).

The history list has a maximum length, called its time-slice. As new events occur, existing events are aged, and the oldest events are forgotten. The time-slice can be changed with the function CHANGESLICE. Larger time-slices enable longer memory spans, but tie up correspondingly greater amounts of storage. Since you seldom need really ancient history, a relatively small time-slice such as 30 events is usually adequate, although some users prefer to set the time-slice as large as 200 events.

Each individual event on *EVENTS* is a list of the form (*INPUT ID VALUE . PROPS*). For Exec events, *ID* is a list (*EVENT-NUMBER EXEC-ID*). The *EVENT-NUMBER* is the number of the event, while the *EXEC-ID* is a string that uniquely identifies the Exec. (The *EXEC-ID* is used to identify which events belong to the "same" Exec.) *VALUE* is the (first) value of the event. *PROPS* is a property list used to associate other information with the event (described below).

INPUT is the input sequence for the event. Normally, this is just the input that you type in. For an APPLY-format input this is a list consisting of two expressions; for an EVAL-format input, this is a list of just one expression; for an input entered as list of atoms, *INPUT* is simply that list. For example,

User Input	*INPUT* is:
LIST(1 2)	(LIST (1 2))
(LIST 1 1)	((LIST 1 1))
DIR "{DSK}<LISPFILES>"*cr*	(DIR "{DSK}<LISPFILES>")

If you type in an Exec command that executes other events (REDO, USE, etc.), several events might result. When there is more than one input, they are wrapped together into one invocation of the DO-EVENTS command.

The same convention is used for representing multiple inputs when a USE command involves sequential substitutions. For example, if you type FBOUNDP(FOO) and then USE

FIE FUM FOR FOO, the input sequence that will be constructed is DO-EVENTS (EVENT FBOUNDP (FIE)) (EVENT FBOUNDP (FUM)), which is the result of substituting FIE for FOO in (FBOUNDP (FOO)) concatenated with the result of substituting FUM for FOO in (FBOUNDP (FOO)).

PROPS is a property list of the form (PROPERTY$_1$ VALUE$_1$ PROPERTY$_2$ VALUE$_2$ ...), that can be used to associate arbitrary information with a particular event. Currently, the following properties are used by the Exec:

SIDE

A list of the side effects of the event. See UNDOSAVE.

LISPXPRINT

Used to record calls to EXEC-FORMAT, and printed by the ?? command.

Making or Changing an Exec

(**XCL:ADD-EXEC** *&KEY PROFILE REGION TTY ID*)   [Function]

Creates a new process and window with an Exec running in it. *PROFILE* is the type of the Exec to be created (see below under XCL:SET-EXEC-TYPE). *REGION* optionally gives the shape and location of the window to be used. If not provided you will be prompted. *TTY* is a flag, which, if true, causes the tty to be given to the new Exec process. *ID* is a string identifier to use for events generated in this exec. *ID* defaults to the number given to the Exec process created.

(**XCL:EXEC** *&KEY WINDOW PROMPT COMMAND-TABLES ENVIRONMENT PROFILE TOP-LEVEL-P TITLE FUNCTION ID*)   [Function]

This is the main entry to the Exec. The arguments are:

WINDOW defaults to the current TTY display stream, or can be provided a window in which the Exec will run.

PROMPT is the prompt to print.

COMMAND-TABLES is a list of hash-tables for looking up commands (e.g., *EXEC-COMMAND-TABLE* or *DEBUGGER-COMMAND-TABLE*).

ENVIRONMENT is a lexical environment used to evaluate things in.

READTABLE is the default readtable to use (defaults to the "Common Lisp" readtable).

PROFILE is a way to set the Exec's type (see above, "Multiple Execs and the Exec's Type").

TOP-LEVEL-P is a boolean, which should be true if this Exec is at the top level (it's NIL for debugger windows, etc).

TITLE is an identifying title for the window title of the Exec.

FUNCTION is a function used to actually evaluate events, default is EVAL-INPUT.

ID is a string identifier to use for events generated in this Exec.   *ID* defaults to the number given to the Exec process.

XCL:*PER-EXEC-VARIABLES*   [Variable]

A list of pairs of the form (*VAR INIT*). Each time an Exec is entered, the variables in **PER-EXEC-VARIABLES** are rebound to the value returned by evaluating *INIT*. The initial value of **PER-EXEC-VARIABLES** is:

```
((*PACKAGE* *PACKAGE*)
 (* *)
 (** **)
 (*** ***)
 (+ +)
 (++ ++)
 (+++ +++)
 (- -)
 (/ /)
 (// //)
 (/// ///)
 (HELPFLAG T)
 (*EVALHOOK* NIL)
 (*APPLYHOOK* nil)
 (*ERROR-OUPUT* *TERMINAL-IO*)
 (*READTABLE* *READTABLE*)
 (*package* *package*)
 (*eval-function* *eval-function*)
 (*exec-prompt* *exec-prompt*)
 (*debugger-prompt* *debugger-prompt*))
```

Most of these cause the values to be (re)bound to their current value in any inferior Exec, or to NIL, their value at the "top level".

XCL:*EVAL-FUNCTION*   [Variable]

Bound to the function used by the Exec to evaluate input.   Typically in an Interlisp Exec this is IL:EVAL, and in a Common Lisp Exec, CL:EVAL.

XCL:*EXEC-PROMPT*   [Variable]

Bound to the string printed by the Exec as a prompt for input.   Typically in an Interlisp Exec this is " ← ", and in a Common Lisp Exec, "> ".

XCL:*DEBUGGER-PROMPT*   [Variable]

Bound to the string printed by the debugger Exec as a prompt for input. Typically in an Interlisp Exec this is " ← : ", and in a Common Lisp Exec, ": ".

(**XCL:EXEC-EVAL** *FORM &OPTIONAL ENVIRONMENT*)   [Function]

Evaluates *FORM* (using EVAL) in the lexical environment *ENVIRONMENT* the same as though it were typed in to EXEC, i.e., the event is recorded, and the evaluation is made undoable by substituting the UNDOABLE-functions for the corresponding destructive functions.   XCL:EXEC-EVAL returns the value(s) of the form, but does not print it, and does not reset the variables *, **, ***, etc.

(**XCL:EXEC-FORMAT** *CONTROL-STRING &REST ARGUMENTS*)　　　　　　　[Function]

> In addition to saving inputs and values, the Exec saves many system messages on the history list. For example, `FILE CREATED ...`, `FN redefined`, `VAR reset`, output of `TIME`, `BREAKDOWN`, `ROOM`, save their output on the history list, so that when `??` prints the event, the output is also printed. The function `XCL:EXEC-FORMAT` can be used in your code similarly. `XCL:EXEC-FORMAT` performs (`APPLY #'CL:FORMAT *TERMINAL-IO* CONTROL-STRING ARGUMENTS`) and also saves the format string and arguments on the history list associated with the current event.

(**XCL:SET-EXEC-TYPE** *NAME*)　　　　　　　　　　　　　　　　[Function]

> Sets the type of the current Exec to that indicated by *NAME*. This can be used to set up the Exec to your liking. *NAME* may be an atom or string. Possible names are:

```
        INTERLISP, IL  *READTABLE* INTERLISP
                       *PACKAGE* INTERLISP
                       XCL:*DEBUGGER-PROMPT* "←: "
                       XCL:*EXEC-PROMPT* "←"
                       XCL:*EVAL-FUNCTION* IL:EVAL

 XEROX-COMMON-LISP, XCL  *READTABLE* XCL
                       *PACKAGE* XCL-USER
                       XCL:*DEBUGGER-PROMPT* ": "
                       XCL:*EXEC-PROMPT* "> "
                       XCL:*EVAL-FUNCTION* CL:EVAL

       COMMON-LISP, CL  *READTABLE* LISP
                       *PACKAGE* USER
                       XCL:*DEBUGGER-PROMPT* ": "
                       XCL:*EXEC-PROMPT* "> "
                       XCL:*EVAL-FUNCTION* CL:EVAL

        OLD-INTERLISP-T  *READTABLE* OLD-INTERLISP-T
                       *PACKAGE* INTERLISP
                       XCL:*DEBUGGER-PROMPT* "←: "
                       XCL:*EXEC-PROMPT* ": "
                       XCL:*EVAL-FUNCTION* IL:EVAL
```

(**XCL:SET-DEFAULT-EXEC-TYPE** *NAME*)　　　　　　　　　　　　[Function]

> Like `XCL:SET-EXEC-TYPE`, but sets the type of Execs created by default, as from the background menu. Initially `XCL`. This can be used in your greet file to set default Execs to your liking.

Editing Exec Input

The Exec features an input editorwhich provides completion, spelling correction, help facility, and character-level editing. The implementation is borrowed from the Interlisp module `TTYIN`. This section describes the use of the `TTYIN` editor from the perspective of the Exec.

Editing Your Input

Some editing operations can be performed using any of several characters; characters that are interrupts will, of course, not be read, so several alternatives are given. The following characters may be used to edit your input:

> **CONTROL-A**

BACKSPACE Deletes a character. At the start of the second or subsequent lines of your input, deletes the last character of the previous line.

CONTROL-W Deletes a "word". Generally this means back to the last space or parenthesis.

CONTROL-Q Deletes the current line, or if the current line is blank, deletes the previous line.

CONTROL-R Refreshes the current line. Two in a row refreshes the whole buffer (when doing multiline input).

ESCAPE Tries to complete the current word from the spelling list USERWORDS. In the case of ambiguity, completes as far as is uniquely determined, or beeps.

UNDO key Retrieves characters from the previous non-empty buffer when it is able to; e.g., when typed at the beginning of the line this command restores the previous line you typed; when typed in the middle of a line fills in the remaining text from the old line; when typed following CONTROL-Q or CONTROL-W restores what those commands erased.

CONTROL-X Goes to the end of your input (or end of expression if there is an excess right parenthesis) and returns if parentheses are balanced.

If you are already at the end of the input and the expression is balanced except for lacking one or more right parentheses, CONTROL-X adds the required right parentheses to balance and returns.

During most kinds of input, lines are broken, if possible, so that no word straddles the end of the line. The pseudo-carriage return ending the line is still read as a space, however; i.e., the program keeps track of whether a line ends in a carriage return or is merely broken at some convenient point. You will not get carriage returns in your strings unless you explicitly type them.

Using the Mouse

Editing with the mouse during TTYIN input is slightly different than with other modules. The mouse buttons are interpreted as follows during TTYIN input:

LEFT Moves the caret to where the cursor is pointing. As you hold down LEFT, the caret moves around with the cursor; after you let up, any type-in will be inserted at the new position.

MIDDLE
LEFT+RIGHT Like LEFT, but moves only to word boundaries.

RIGHT Deletes text from the caret to the cursor, either forward or backward. While you hold down RIGHT, the text to be deleted is inverted; when you let up, the text goes away. If you let up outside the scope of the text, nothing is deleted (this is how to cancel this operation).

If you hold down MOVE, COPY, SHIFT or CTRL while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. The selection is made by holding the appropriate key down while pressing the mouse buttons LEFT (to select a character) or MIDDLE (to select a word), and optionally extend the selection either left or right using RIGHT. While you are doing this, the caret does not move, but the selected text is highlighted in a manner indicating what is about to happen. When the selection is complete, release the mouse buttons and then lift up on MOVE/COPY/CTRL/SHIFT and the appropriate action will occur:

COPY **SHIFT**	The selected text is inserted as if it were typed. The text is highlighted with a broken underline during selection.
CTRL	The selected text is deleted. The text is complemented during selection.
MOVE **CTRL**+**SHIFT**	Combines copy and delete. The selected text is moved to the caret.

You can cancel a selection in progress by pressing LEFT or MIDDLE as if to select, and moving outside the range of the text.

The most recent text deleted by mouse command can be inserted at the caret by typing the UNDO key. This is the same key that retrieves the previous buffer when issued at the end of a line.

Editing Commands

A number of characters have special effects while typing to the Exec. Some of them merely move the caret inside the input stream. While caret positioning can often be done more conveniently with the mouse, some of the commands, such as the case changing commands, can be useful for modifying the input.

In the descriptions below, current word means the word the cursor is under, or if under a space, the previous word.   Currently, parentheses are treated as spaces, which is usually what you want, but can occasionally cause confusion in the word deletion commands.

Most commands can be preceded by a numeric argument. A numeric argument can be a number or an escape. You enter the numeric argument by holding down the meta key and entering a number. You only need to hold down the meta key for the firs digit of the argument. Entering escape as a numeric argument means infinity.

Some commands also accept negative arguments, but some only look at the magnitude of the argument. Most of these commands are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands

Meta-BACKSPACE	Backs up one (or n) characters.
Meta-SPACE	Moves forward one (or n) characters.
Meta-^	Moves up one (or n) lines.
Meta-LINEFEED	Moves down one (or n) lines.
Meta-(	Moves back one (or n) words.
Meta-)	Moves ahead one (or n) words.
Meta-tab	Moves to end of line; with an argument moves to nth end of line; **Meta-Control-tab** goes to end of buffer.
Meta-Control-L	Moves to start of line (or nth previous, or start of buffer).
Meta-{	Goes to start of buffer.
Meta-}	Goes to end of buffer.

Meta-[Moves to beginning of the current list, where cursor is currently under an element of that list or its closing paren. (See also the auto-parenthesis-matching feature below under "Assorted Flags".)

Meta-] Moves to end of current list.

Meta-Sx Skips ahead to next (or nth) occurrence of character x, or rings the bell.

Meta-Bx Backward search.

Buffer Modification Commands

Meta-Zx Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command.

Meta-A
Meta-R Repeats the last S, B, or Z command, regardless of any intervening input.

Meta-K Kills the character under the cursor, or n chars starting at the cursor.

Meta-CR When the buffer is empty is the same as undo i.e. restores buffer's previous contents. Otherwise is just like a <cr> (except that it also terminates an insert). Thus, **Meta-CR Meta-CR** will repeat the previous input (as will undo<cr> without the meta key).

Meta-O Does "Open line", inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.

Meta-T Transposes the characters before and after the cursor. When typed at the end of a line, transposes the previous two characters. Refuses to handle odd cases, such as tabs.

Meta-G Grabs the contents of the previous line from the cursor position onward. **Meta-n Meta-G** grabs the nth previous line.

Meta-L Puts the current word, or n words on line, in lower case. **Meta-<escape> Meta-L** puts the rest of the linein lower case; or if given at the end of line puts the entire line in lower case.

Meta-U Analogous to **Meta-L**, for putting word, line, or portion of line in upper case.

Meta-C Capitalizes. If you give it an argument, only the first word is capitalized; the rest are just lowercased.

Meta-Control-Q Deletes the current line. **Meta-<escape> Meta-Control-Q** deletes from the current cursor position to the end of the buffer. No other arguments are handled.

Meta-Control-W Deletes the current word, or the previous word if sitting on a space.

Miscellaneous Commands

Meta-P Prettyprints buffer. Clears the buffer and reprints it using prettyprint. If there are not enough right parentheses, it will supply more; if there are too many, any excess remains unprettyprinted at the end of the buffer. May refuse to do anything if there is an unclosed string or other error trying to read the buffer.

Meta-N Refreshes line. Same as **Control-R**. **Meta-**<escape> **Meta-N** refreshes the whole buffer; **Meta-**n **Meta-N** refreshes n lines. Cursor movement in `TTYIN` depends on `TTYIN` being the only source of output to the window; in some circumstances, you may need to refresh the line for best results.

Meta-Control-Y Gets an Interlisp Exec. **Meta-**<escape> **Meta-Control-Y** Gets an Interlisp Exec, but first unreads the contents of the buffer from the cursor onward. Thus if you typed at `TTYIN` something destined for Interlisp, you can do **Meta-Control-L Meta-**<escape> **Meta-Control-Y** and give it to Lisp.

Meta-_ Adds the current word to the spelling list `USERWORDS`. With zero argument, removes word. See `TTYINCOMPLETEFLG` .

Useful Macros

If the event is considered short enough, the Exec command `FIX` will load the buffer with the event's input, rather than calling the structure editor. If you really wanted the Lisp editor for your fix, you can say `FIX EVENT - |TTY:|` .

?= Handler

Typing the characters `?=`<cr> displays the arguments to the function currently in progress. Since `TTYIN` wants you to be able to continue editing the buffer after a `?=`, it prints the arguments below your type-in and then puts the cursor back where it was when `?=` was typed.

Assorted Flags

These flags control aspects of `TTYIN`'s behavior. Some have already been mentioned. All are initially set to `T`.

?ACTIVATEFLG [Variable]

If true, enables the feature whereby `?` lists alternative completions from the current spelling list.

SHOWPARENFLG [Variable]

If true, then whenever you are typing Lisp input and type a right parenthesis, `TTYIN` will briefly move the cursor to the matching parenthesis, assuming it is still on the screen. The cursor stays there for about 1 second, or until you type another character (i.e., if you type fast you will never notice it).

USERWORDS [Variable]

`USERWORDS` contains words you mentioned recently: functions you have defined or edited, variables you have set or evaluated at the executive level, etc. This happens to be a very convenient list for context-free escape completion; if you have recently edited a function, chances are good you may want to edit it again (typing "`ED(xx$)`") or type a call to it. If there is no completion for the current word from `USERWORDS`, or there is more than one possible completion, `TTYIN` beeps. If typed when not inside a word, Escape completes to the value of `LASTWORD`, i.e., the last thing you typed that the Exec noticed,

except that Escape at the beginning of the line is left alone (it is an Old Interlisp Exec command).

If you really wanted to enter an escape, you can, of course, just quote it with a CONTROL-V, like you can other control characters.

You may explicitly add words to USERWORDS yourself that would not get there otherwise. To make this convenient online the edit command [←] means "add the current atom to USERWORDS" (you might think of the command as pointing out this atom). For example, you might be entering a function definition and want to point to one or more of its arguments or prog variables. Giving an argument of zero to this command will instead remove the indicated atom from USERWORDS.

Note that this feature loses some of its value if the spelling list is too long, if there are too many alternative completions for you to get by with typing a few characters followed by escape. Lisp's maintenance of the spelling list USERWORDS keeps the temporary section (which is where everything goes initially unless you say otherwise) limited to \\#USERWORDS atoms, initially 100. Words fall off the end if they haven't been used (they are used if FIXSPELL corrects to one, or you use <escape> to complete one).

Old Interlisp T compatibility

The Old Interlisp exec contains a few extra Exec commands not listed above. They are explained here.

In addition to the normal Event addresses you can also specify the following Event addresses:

= Specifies that the next object is to be searched for in the values of events, instead of the inputs

SUCHTHAT PRED Specifies an event for which the function PRED returns true. PRED should be a function of two arguments, the input portion of the event, and the event itself.

PAT Any other event address command specifies an event whose input contains an expression that matches PAT. When multiple Execs are active, all events are searched, no matter which Exec they belong to. The pattern can be a simple symbol, or a more complex search pattern.

Significant Changes in MEDLEY Rele███████████████████████████████ase

There are two major differences between the Medley release and older versions of the system:

- SETQ does not interact with the File Manager. In older releases (Koto, etc.), when you typed in (SETQ FOO some-new-value) the executive responded with (FOO reset) and the file manager was told that FOO's value had changed. Files containing FOO were marked for cleanup, if none existed you were prompted for one when you typed (FILES?).

This is still the case in the Old Interlisp executive but not in any of the others. If you are setting a variable that is significant to a program and you want to save it on a file, you should use the Common Lisp macro CL:DEFPARAMETER instead of SETQ. This will give the symbol a definition of type VARIABLES (instead of VARS), and it will be noticed by the File Manager. Subseqent

changes to the variable must be done by another call to `CL:DEFPARAMETER` or by editing it using `ED` (not `DV`).

- The following functions and variables are only available in the Old Interlisp Exec: `LISPX`, `USEREXEC`, `LISPXEVAL`, `READBUF`, `(READLINE)`, `(LISPXREAD)`, `(LISPXREADP)`, `(LISPXUNREAD)`, `(PROMPTCHAR)`, `(HISTORYSAVE)`, `(LISPXSTOREVALUE)`, `(LISPXFIND)`, `(HISTORYFIND)`, `(HISROTYMATCH)`, `(ENTRY)`, `(UNDOSAVE)`, `#UNDOSAVES`, `(NEW/FN)`, `(LISPX/)`,`(UNDOLISPX)`,`(UNDOLISPX1)`, and `(PRINTHISTORY)`.

The function `USEREXEC` invokes an old-style executive, but uses the package and readtable of its caller. Callers of `LISPXEVAL` should use `EXEC-EVAL` instead.

Occasionally, while a program is running, an error occurs which stops the computation. Errors can be caused in different ways. A coding mistake may have caused the wrong arguments to be passed to a function, or caused the function to attempt something illegal. For example, PLUS will cause an error if its arguments are not numbers. It is also possible to interrupt a computation by typing one of the "interrupt characters," such as Control-D or Control-E (Medley interrupt characters are listed in Chapter 30). Finally, you can specify that certain functions automatically cause an error whenever they are entered (see Chapter 15). This facilitates debugging by allowing you to examine the context within the computation.

When an error occurs, the system can either reset and unwind the stack, or go into a "break", and attempt to debug the program. You can modify the mechanism that decides whether to unwind the stack or break, and is described in the Controlling When to Break section in this chapter. Within a break, Medley offers an extensive set of "break commands".

This chapter explains what happens when errors occur. It also tells you how to handle program errors using breaks and break commands. The debugging capabilities of the break window facility are described, as well as the variables that control its operation. Finally, advanced facilities for modifying and extending the error mechanism are presented.

Breaks

One of the most useful debugging facilities in Medley is the ability to put the system into a "break", stopping a computation at any point, allowing you to interrogate the state of the world and affect the course of the computation. When a break occurs, a "break window" (see the Break Windows section below) is brought up near the TTY window of the broken process. The break window looks like a top-level executive window, except that the prompt character is "**:**" instead of "←" as in the top-level executive. A break saves the environment where the break occurred, so that you may evaluate variables and expressions in the borken environment. In addition, the break program recognizes a number of useful "break commands", providing an easy way to interrogate the state of the broken computation.

Breaks may be entered in several ways. Some interrupt characters (Chapter 30) automatically cause a break whenever you type them. Function errors may also cause a break, depending on the depth of the computation (see Controlling When to Break below). Finally, Medley provides facilities which make it easy to "break" suspect functions so that they always cause a break whenever they are entered.

Within a break you have access to all of the power of Medley; you can do anything you can do at the top-level executive. For example, you can evaluate an expression, call the editor, change the function, and evaluate the expression again, all without leaving the break. You can also type in commands like REDO, and UNDO (Chapter 13), to redo or undo previously executed events, including break commands.

Similarly, you can prettyprint functions, define new functions or redefine old ones, load a file, compile functions, time a computation, etc. In addition, you can examine the stack (see Chapter 11), and even force a return back to some higher function via the functions RETFROM or RETEVAL.

Once a break occurs, *you* are in complete control of the flow of the computation, and the computation will not proceed without specific instruction from you. If you type in an expression whose evaluation

causes an error, the break is maintained. Similarly if you abort a computation initiated from within the break (by typing Control-E), the break is maintained. Only if you give one of the commands that exits from the break, or evaluates a form which does a `RETFROM` or `RETEVAL` out of `BREAK1`, will the computation continue. Also, `BREAK1` does not "turn off" Control-D, so a Control-D will force an immediate return to the top level.

Break Windows

When a break occurs, a break window is brought up near the TTY window of the borken process and the terminal stream switched to it. The title of the break window is changed to the name of the broken function and the reason for the break. If a break occurs under a previous break, a new break window is created.

If a break is caused by a storage full error, the display break package will not try to open a new break window, since this would cause an infinite loop.

While in a break window, clicking the middle button brings up a menu of break commands: `EVAL`, `EDIT`, `revert`, ↑, `OK`, `BT`, `BT!`, and `?=`. Clicking on these commands is equivalent to typing the corresponding break commandm, except `BT` and `BT!` which behave differently from the typed-in commands (see Break Commands below).

The `BT` and `BT!` menu commands bring up a backtrace menu beside the break window showing the frames on the stack. `BT` shows frames for which `REALFRAMEP` is `T`; `BT!` shows all frames. When one of the frames is selected from the backtrace menu, it is grayed and the function name and the variables bound in that frame (including local variables and `PROG` variables) are printed in the "backtrace frame window." If the left button is used for the selection, only named variables are printed. If the middle button is used, all variables are printed (variables without names appear as `*var*` *N*). The "backtrace frame" window is an inspect window (see Chapter 26). In this window, the left button is used to select the name of the function, the names of the variables or the values of the variables. For example, below is a picture of a break window with a backtrace menu created by `BT`. The `OPENSTREAM` stack frame has been selected, so its variables are shown in an inspect window on top of the break window:

After selecting an item, the middle button brings up a menu of commands that apply to the selected item. If the function name is selected, you are given a choice of editing the function or seeing the compiled code with `INSPECTCODE` (Chapter 26). If you edit the function in this way, the editor is called in the broken process, so variables evaluated in the editor are in the broken process.

If a variable name is selected, the command SET is offered. Selecting SET will READ a value and set the selected to the value read.

Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. Exercise caution in setting variables in other than your own code.

If a value is selected, the inspector is called on the selected value.

The internal break variable LASTPOS (see the section below) is set to the selected backtrace menu frame so that the normal break commands EDIT, revert, and ?= work on the currently selected frame. The commands EVAL, revert, ↑, OK, and ?= in the break menu cause the corresponding commands to be "typed in." This means that these break commands will not have the intended effect if characters have already been typed in. The typed-in break commands BT, BTV, etc. use the value of LASTPOS to determine where to start listing the stack, so selecting a stack frame name in the backtrace menu affects these commands.

Break Commands

The basic function of the break package is BREAK1. BREAK1 is just another Interlisp function, not a special system feature like the interpreter or the garbage collector. It has arguments, and returns a value, the same as any other function. For more information on the function BREAK1, see Creating Breaks with BREAK1 below.

The value returned by BREAK1 is called "the value of the break." You can specify this value explicitly by using the RETURN break command (see below). But in most cases, the value of a break is given implicitly, via a GO or OK command, and is the result of evaluating "the break expression." The break expression, stored in the variable BRKEXP, is an expression equivalent to the computation that would have taken place had no break occurred. For example, if you break on the function FOO, the break expression is the body of the definition of FOO. When you type OK or GO, the body of FOO is evaluated, and its value returned as the value of the break, i.e., to whatever function called FOO. BRKEXP is set up by the function that created the call to BREAK1. For functions broken with BREAK or TRACE, BRKEXP is equivalent to the body of the definition of the broken function (see Chapter 15). For functions broken with BREAKIN, using BEFORE or AFTER, BRKEXP is NIL. For BREAKIN AROUND, BRKEXP is the indicated expression (see Chapter 15).

BREAK1 recognizes a large set of break commands. These are typed in *without* parentheses. In order to facilitate debugging of programs that perform input operations, the carriage return that is typed to complete the GO, OK, EVAL, etc. commands is discarded by BREAK1, so that it will not be part of the input stream after the break.

GO [Break Command]

Evaluates BRKEXP, prints its value, and returns it as the value of the break. Releases the break and allows the computation to proceed.

OK [Break Command]

Same as GO except that the value of BRKEXP is not printed.

EVAL [Break Command]

Same as `OK` except that the break is maintained after the evaluation. The value of `EVAL` is bound to the local variable `!VALUE`, which you can interrogate. Typing `GO` or `OK` following `EVAL` will not cause `BRKEXP` to be reevaluated, but simply returns the value of `!VALUE` as the value of the break. Typing another `EVAL` will cause reevaluation. `EVAL` is useful when you are not sure whether the break will produce the correct value and want to examine it before continuing with the computation.

RETURN *FORM* [Break Command]

FORM is evaluated, and returned as the value of the break. For example, one could use the `EVAL` command and follow this with `RETURN (REVERSE !VALUE)`.

↑ [Break Command]

Calls `ERROR!` and aborts the break, making it "go away" without returning a value. This is a useful way to unwind to a higher level break. All other errors, including those encountered while executing the `GO`, `OK`, `EVAL`, and `RETURN` commands, maintain the break.

The following four commands refer to "the broken function", whose name is stored in the `BREAK1` argument `BRKFN`.

!GO [Break Command]

The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited with the value printed.

!OK [Break Command]

The broken function is unbroken, the break expression is evaluated, the function is rebroken, and then the break is exited without the value printed.

UB [Break Command]

Unbreaks the broken function.

@ [Break Command]

Resets the variable `LASTPOS`, which establishes a context for the commands `?=`, `ARGS`, `BT`, `BTV`, `BTV*`, `EDIT`, and `IN?` described below. `LASTPOS` is the position of a function call on the stack. It is initialized to the function just before the call to `BREAK1`, i.e., `(STKNTH -1 'BREAK1)`.

When control passes from `BREAK1`, e.g. as a result of an `EVAL`, `OK`, `GO`, `REVERT`, `↑` command, or via a `RETFROM` or `RETEVAL` you type in, `(RELSTK LASTPOS)` is executed to release this stack pointer.

`@` treats the rest of the teletype line as its argument(s). It first resets `LASTPOS` to `(STKNTH -1 'BREAK1)` and then for each atom on the line, `@` searches down the stack for a call to that atom. The following atoms are treated specially:

@ Do not reset LASTPOS to (STKNTH -1 'BREAK1) but leave it as it was, and continue searching from that point.

a number *N* If negative, move LASTPOS down the stack *N* frames. If positive, move LASTPOS up the stack *N* frames.

/ The next atom on the line (which should be a number) specifies that the *previous* atom should be searched for that many times. For example, "@ FOO / 3" is equivalent to "@ FOO FOO FOO".

= Resets LASTPOS to the *value* of the next expression, e.g., if the value of FOO is a stack pointer, "@ = FOO FIE" will search for FIE in the environment specified by (the value of) FOO.

For example, if the push-down stack looks like:

```
[9]    BREAK1
[8]    FOO
[7]    COND
[6]    FIE
[5]    COND
[4]    FIE
[3]    COND
[2]    FIE
[1]    FUM
```

then "@ FIE COND" will set LASTPOS to the position corresponding to *[5]*; "@ @ COND" will then set LASTPOS to *[3]*; and "@ FIE / 3 – 1" to *[1]*.

If @ cannot successfully complete a search for function *FN*, it searches the stack again from that point looking for a call to a function whose name is a possible misspelling of *FN* (see spelling correction in Chapter 20). If the search is still unsuccessful, @ types (*FN* NOT FOUND), and then aborts.

When @ finishes, it types the name of the function at LASTPOS, i.e., (STKNAME LASTPOS).

@ can be used on BRKCOMS (see Creating Breaks with BREAK1 below). In this case, the *next* command on BRKCOMS is treated the same as the rest of the teletype line.

?=   [Break Command]

This is a multi-purpose command. Its most common use is to interrogate the value(s) of the arguments of the broken function. For example, if FOO has three arguments (X Y Z), then typing ?= to a break on FOO will produce:

```
:?=
X = value of X
Y = value of Y
Z = value of Z
:
```

`?=` operates on the rest of the teletype line as its arguments. If the line is empty, as in the above case, it operates on all of the arguments of the broken function. If the you type `?=` X `(CAR Y)`, you will see the value of X, and the value of `(CAR Y)`. The difference between using `?=` and typing X and `(CAR Y)` directly to `BREAK1` is that `?=` evaluates its inputs as of the stack frame `LASTPOS`, i.e., it uses `STKEVAL`. This provides a way of examining variables or performing computations *as of a particular point on the stack.* For example, `@ FOO / 2` followed by `?= X` will allow you to examine the value of X in the previous call to `FOO`, etc.

`?=` also recognizes numbers as referring to the correspondingly numbered argument, i.e., it uses `STKARG` in this case. Thus

```
:@ FIE
FIE
:?= 2
```

will print the name and value of the second argument of `FIE`.

`?=` can also be used on `BRKCOMS` (see Creating Breaks with BREAK1 below), in which case the next command on `BRKCOMS` is treated as the rest of the teletype line. For example, if `BRKCOMS` is `(EVAL ?= (X Y) GO)`, `BRKEXP` is evaluated, the values of X and Y printed, and then the function exited with its value being printed.

`?=` prints variable values using the function `SHOWPRINT` (see Chapter 25), so that if `SYSPRETTYFLG = T`, the value is prettyprinted.

`?=` is a universal mnemonic for displaying argument names and their corresponding values. In addition to being a break command, `?=` is an edit macro that prints the argument names and values for the current expression (see Chapter 16), and a read macro (actually `?` is the read macro character) which does the same for the current level list being read.

PB   [Break Command]

Prints the bindings of a given variable. Similar to `?=`, except ascends the stack starting from `LASTPOS`, and, for each frame in which the given variable is bound, prints the frame name and value of the variable (with `PRINTLEVEL` reset to `(2 . 3)`), e.g.

```
:PB FOO
@  FN1:  3
@  FN2:  10
@  TOP:  NOBIND
```

`PB` is also a programmer's assistant command (see Chapter 13) that can be used when not in a break. `PB` is implemented via the function `PRINTBINDINGS`.

BT   [Break Command]

Prints a backtrace of function names starting at `LASTPOS`. The value of `LASTPOS` is changed by selecting an item from the backtrace menu (see the Break Window Variables section below) or by the `@` command. The several nested calls in system packages such as break, edit, and the top level executive appear as the single entries `**BREAK**`, `**EDITOR**`, and `**TOP**` respectively.

BTV [Break Command]

> Prints a backtrace of function names *with* variables beginning at LASTPOS.

> The value of each variable is printed with the function SHOWPRINT (see Chapter 25), so that if SYSPRETTYFLG = T, the value is prettyprinted.

BTV+ [Break Command]

> Same as BTV except also prints local variables and arguments to SUBRs.

BTV* [Break Command]

> Same as BTV except prints arguments to local variables.

BTV! [Break Command]

> Same as BTV except prints *everything* on the stack.

BT, BTV, BTV+, BTV*, and BTV! all take optional functional arguments. Use these arguments to choose functions to be *skipped* on the backtrace. As the backtrace scans down the stack, the name of each stack frame is passed to each of the arguments of the backtrace command. If any of these functions returns a non-NIL value, then that frame is skipped, and not shown in the backtrace. For example, BT EXPRP will skip all functions definied by expr definitions, BTV (LAMBDA (X) (NOT (MEMB X FOOFNS))) will skip all but those functions on FOOFNS. If used on BRKCOMS (see Creating Breaks with BREAK1 below) the functional argument is no longer optional, i.e., the next element on BRKCOMS must either be a list of functional arguments, or NIL if no functional argument is to be applied.

For BT, BTV, BTV+, BTV*, and BTV!, if Control-P is used to change a printlevel during the backtrace, the printlevel is restored after the backtrace is completed.

The value of BREAKDELIMITER, initially the carriage return character, is printed to delimit the output of ?= and backtrace commands. You can reset it (e.g. to a comma) for more linear output.

ARGS [Break Command]

> Prints the names of the variables bound at LASTPOS, i.e., (VARIABLES LASTPOS) (see Chapter 11). For most cases, these are the arguments to the function entered at that position, i.e., (ARGLIST (STKNAME LASTPOS)).

REVERT [Break Command]

> Goes back to position LASTPOS on stack and reenters the function called at that point with the arguments found on the stack. If the function is not already broken, REVERT first breaks it, and then unbreaks it after it is reentered.

> REVERT can be given the position using the conventions described for @, e.g., REVERT FOO -1 is equivalent to @ FOO -1 followed by REVERT.

> REVERT is useful for restarting a computation in the situation where a bug is discovered at some point *below* where the problem actually occurred. REVERT essentially says "go back there and start over in a break." REVERT will work correctly if the names or arguments to the function, or even its function type, have been changed.

`ORIGINAL` [Break Command]

> For use in conjunction with `BREAKMACROS` (see Creating Breaks with BREAK1 below).
> Form is `(ORIGINAL  .  COMS)`. `COMS` are executed without regard for `BREAKMACROS`.
> Useful for redefining a break command in terms of itself.

`EDIT` [Break Command]

> Designed for use in conjunction with breaks caused by errors. Facilitates editing the
> expression causing the break:

```
NON-NUMERIC ARG
NIL
(IPLUS BROKEN)
:EDIT
IN FOO...
(IPLUS X Z)
EDIT
*(3 Y)
*OK
FOO
:
```

> and you can continue by typing `OK`, `EVAL`, etc.

> This command is very simple conceptually, but its implementation is complicated by all of
> the exceptional cases involving interactions with compiled functions, breaks on user
> functions, error breaks, breaks within breaks, et al. Therefore, we shall give the following
> simplified explanation which will account for 90% of the situations arising in actual usage.
> For those others, `EDIT` will print an appropriate failure message and return to the break.

> `EDIT` begins by searching up the stack beginning at `LASTPOS` (set by `@` command, initially
> position of the break) looking for a form, i.e., an internal call to `EVAL`. Then `EDIT`
> continues from that point looking for a call to an interpreted function, or to `EVAL`. It then
> calls the editor on either the `EXPR` or the argument to `EVAL` in such a way as to look for an
> expression `EQ` to the form that it first found. It then prints the form, and permits
> interactive editing to begin. You can then type successive `0`'s to the editor to see the chain
> of superforms for this computation.

> If you exit from the edit with an `OK`, the break expression is reset, if possible, so that you
> can continue with the computation by simply typing `OK`. (Evaluating the new `BRKEXP`
> will involve reevaluating the form that causes the break, so that if `(PUTD (QUOTE
> (FOO)) BIG-COMPUTATION)` were handled by `EDIT`, *BIG-COMPUTATION* would be
> reevaluated.) However, in some situations, the break expression cannot be reset. For
> example, if a compiled function `FOO` incorrectly called `PUTD` and caused the error `Arg
> not atom` followed by a break on `PUTD`, `EDIT` might be able to find the form headed by
> `FOO`, and also find *that* form in some higher interpreted function. But after you corrected
> the problem in the `FOO`-form, if any, you would still not have informed `EDIT` what to do
> about the immediate problem, i.e., the incorrect call to `PUTD`. However, if `FOO` were
> *interpreted*, `EDIT` would find the `PUTD` form itself, so that when you corrected that form,
> `EDIT` could use the new corrected form to reset the break expression.

IN? [Break Command]

> Similar to EDIT, but just prints parent form, and superform, but does not call the editor, e.g.,
>
> ```
>     ATTEMPT TO RPLAC NIL
>     T
>     (RPLACD BROKEN)
>     :IN?
>     FOO:   (RPLACD X Z)
> ```
>
> Although EDIT and IN? were designed for error breaks, they can also be useful for user breaks. For example, if upon reaching a break on his function FOO, you determine that there is a problem in the *call* to FOO, you can edit the calling form and reset the break expression with one operation by using EDIT.

Controlling When to Break

When an error occurs, the system has to decide whether to reset and unwind the stack, or go into a break. In the middle of a complex computation, it is usually helpful to go into a break, so that you may examine the state of the computation. However, if the computation has only proceeded a little when the error occurs, such as when you mistype a function name, you would normally just terminate a break, and it would be more convenient for the system to simply cause an error and unwind the stack in this situatuation. The decision over whether or not to induce a break depends on the depth of computation, and the amount of time invested in the computation. The actual algorithm is described in detail below; suffice it to say that the parameters affecting this decision have been adjusted empirically so that trivial type-in errors do not cause breaks, but deep errors do.

(**BREAKCHECK** *ERRORPOS ERXN*) [Function]

> BREAKCHECK is called by the error routine to decide whether or not to induce a break when a error occurs. *ERRORPOS* is the stack position at which the error occurred; *ERXN* is the error number. Returns T if a break should occur; NIL otherwise.
>
> BREAKCHECK returns T (and a break occurs) if the "computation depth" is greater than or equal to HELPDEPTH. HELPDEPTH is initially set to 7, arrived at empirically by taking into account the overhead due to LISPX or BREAK.
>
> If the depth of the computation is less than HELPDEPTH, BREAKCHECK next calculates the length of time spent in the computation. If this time is greater than HELPTIME milliseconds, initially set to 1000, then BREAKCHECK returns T (and a break occurs), otherwise NIL.
>
> BREAKCHECK determines the "computation depth" by searching back up the stack looking for an ERRORSET frame (ERRORSETs indicate how far back unwinding is to take place when an error occurs, see the Catching Errors section below). At the same time, it counts the number of internal calls to EVAL. As soon as the number of calls to EVAL exceeds HELPDEPTH, BREAKCHECK immediately stops searching for an ERRORSET and returns T. Otherwise, BREAKCHECK continues searching until either an ERRORSET is found or the top of the stack is reached. (If the second argument to ERRORSET is INTERNAL, the ERRORSET is ignored by BREAKCHECK during this search.) BREAKCHECK then counts the number of function calls between the error and the last ERRORSET, or the top of the stack.

The number of function calls plus the number of calls to `EVAL` (already counted) is used as the "computation depth".

`BREAKCHECK` determines the computation time by subtracting the value of the variable `HELPCLOCK` from the value of (`CLOCK` `2`), the number of milliseconds of compute time (see Chapter 12). `HELPCLOCK` is rebound to the current value of (`CLOCK` `2`) for each computation typed in to `LISPX` or to a break. The time criterion for breaking can be suppressed by setting `HELPTIME` to `NIL` (or a very big number), or by setting `HELPCLOCK` to `NIL`. Setting `HELPCLOCK` to `NIL` will not have any effect beyond the current computation, because `HELPCLOCK` is rebound for each computation typed in to `LISPX` and `BREAK`.

You can suppress all error breaks by setting the top level binding of the variable `HELPFLAG` to `NIL` using `SETTOPVAL` (`HELPFLAG` is bound as a local variable in `LISPX`, and reset to the global value of `HELPFLAG` on every `LISPX` line, so just `SETQ`ing it will not work.) If `HELPFLAG` = `T` (the initial value), the decision whether to cause an error or break is decided based on the computation time and the computation depth, as described above. Finally, if `HELPFLAG` = `BREAK!`, a break will always occur following an error.

Break Window Variables

The appearance and use of break windows is controlled by the following variables:

(**WBREAK** *ONFLG*)   [Function]

If *ONFLG* is non-`NIL`, break windows and trace windows are enabled. If *ONFLG* is `NIL`, break windows are disabled (break windows do not appear, but the executive prompt is changed to "**:**" to indicate that the system is in a break). `WBREAK` returns `T` if break windows are currently enabled; `NIL` otherwise.

MaxBkMenuWidth   [Variable]
MaxBkMenuHeight   [Variable]

The variables `MaxBkMenuWidth` (default 125) and `MaxBkMenuHeight` (default 300) control the maximum size of the backtrace menu. If this menu is too small to contain all of the frames in the backtrace, it is made scrollable in both vertical and horizontal directions.

AUTOBACKTRACEFLG   [Variable]

This variable controls when and what kind of backtrace menu is automatically brought up. The value of `AUTOBACKTRACEFLG` can be one of the following:

NIL   The backtrace menu is not automatically brought up (the default).

T   On error breaks the `BT` menu is brought up.

BT!   On error breaks the `BT!` menu is brought up.

ALWAYS   The `BT` menu is brought up on both error breaks and user breaks (calls to functions broken by `BREAK`).

ALWAYS!   On both error breaks and user breaks the `BT!` menu is brought up.

BACKTRACEFONT   [Variable]

> The backtrace menu is printed in the font BACKTRACEFONT.

CLOSEBREAKWINDOWFLG   [Variable]

> The system normally closes break windows after the break is exited. If CLOSEBREAKWINDOWFLG is NIL, break windows will not be closed on exit. In this case, you must close all break windows.

BREAKREGIONSPEC   [Variable]

> Break windows are positioned near the TTY window of the broken process, as determined by the variable BREAKREGIONSPEC. The value of this variable is a region (see Chapter 27) whose LEFT and BOTTOM fields are an offset from the LEFT and BOTTOM of the TTY window. The WIDTH and HEIGHT fields of BREAKREGIONSPEC determine the size of the break window.

TRACEWINDOW   [Variable]

> The trace window, TRACEWINDOW, is used for tracing functions. It is brought up when the first tracing occurs and stays up until you close it. TRACEWINDOW can be set to a particular window to cause the tracing formation to print there.

TRACEREGION   [Variable]

> The trace window is first created in the region TRACEREGION.

Creating Breaks with BREAK1

The basic function of the break package is BREAK1, which creates a break. A break appears to be a regular executive, with the prompt "**:**", but BREAK1 also detects and interpretes break commands (see the Break Commands section above).

(**BREAK1** *BRKEXP BRKWHEN BRKFN BRKCOMS BRKTYPE ERRORN*)   [NLambda Function]

> If *BRKWHEN* (evaluated) is non-NIL, a break occurs and commands are then taken from *BRKCOMS* or the terminal and interpreted. All inputs not recognized by BREAK1 are simply passed on to the programmer's assistant.

> If *BRKWHEN* is NIL, *BRKEXP* is evaluated and returned as the value of BREAK1, without causing a break.

> When a break occurs, if *ERRORN* is a list whose CAR is a number, ERRORMESS (see the Signalling Errors section below) is called to print an identifying message. If *ERRORN* is a list whose CAR is not a number, ERRORMESS1 (see the Signalling Errors section below) is called. Otherwise, no preliminary message is printed. Following this, the message (*BRKFN* broken) is printed.

> Since BREAK1 itself calls functions, when one of these is broken, an infinite loop would occur. BREAK1 detects this situation, and prints Break within a break on *FN*, and then simply calls the function without going into a break.

The commands `GO`, `!GO`, `OK`, `!OK`, `RETURN` and ↑ are the only ways to leave `BREAK1`. The command `EVAL` causes *BRKEXP* to be evaluated, and saves the value on the variable `!VALUE`. Other commands can be defined for `BREAK1` via `BREAKMACROS` (below).

BRKTYPE is `NIL` for user breaks, `INTERRUPT` for Control-H breaks, and `ERRORX` for error breaks. For breaks when *BRKTYPE* is not `NIL`, `BREAK1` will clear and save the input buffer. If the break returns a value (i.e., is not aborted via ↑ or Control-D) the input buffer is restored.

The fourth argument to `BREAK1` is *BRKCOMS*, a list of break commands that `BREAK1` interprets and executes as though they were keyboard input. One can think of *BRKCOMS* as another input file which always has priority over the keyboard. Whenever *BRKCOMS* = `NIL`, `BREAK1` reads its next command from the keyboard. Whenever *BRKCOMS* is non-`NIL`, `BREAK1` takes `(CAR BRKCOMS)` as its next command and sets *BRKCOMS* to `(CDR BRKCOMS)`. For example, suppose you wished to see the value of the variable `X` *after* a function was evaluated. You could set up a break with *BRKCOMS* = `(EVAL (PRINT X) OK)`, which would have the desired effect. If *BRKCOMS* is non-`NIL`, the value of a break command is not printed. If you desire to see a value, you must print it yourself, as in the above example. The function `TRACE` (see Chapter 15) uses *BRKCOMS*: it sets up a break with two commands; the first one prints the arguments of the function, or whatever you specify, and the second is the command `GO`, which causes the function to be evaluated and its value printed.

Note: If an error occurs while interpreting the *BRKCOMS* commands, *BRKCOMS* is set to `NIL`, and a full interactive break occurs.

The break package has a facility for redirecting ouput to a file. All output resulting from *BRKCOMS* is output to the value of the variable `BRKFILE`, which should be the name of an open file. Output due to user type-in is not affected, and will always go to the terminal. `BRKFILE` is initially `T`.

BREAKMACROS   [Variable]

BREAKMACROS is a list of the form `((`*NAME₁*` `*COM1₁*` ... `*COM1ₙ*`)(`*NAME₂*` `*COM2₁*` ... `*COM2ₙ*`)...)`. Whenever an atomic command is given to `BREAK1`, it first searches the list `BREAKMACROS` for the command. If the command is equal to *NAMEᵢ*, `BREAK1` simply appends the corresponding commands to the front of *BRKCOMS*, and goes on. If the command is not found on `BREAKMACROS`, `BREAK1` then checks to see if it is one of the built in commands, and finally, treats it as a function or variable as before.

If the command is not the name of a defined function, bound variable, or `LISPX` command, `BREAK1` will attempt spelling correction using `BREAKCOMSLST` as a spelling list. If spelling correction is unsuccessful, `BREAK1` will go ahead and call `LISPX` anyway, since the atom may also be a misspelled history command.

For example, the command `ARGS` could be defined by including on `BREAKMACROS` the form:

```
(ARGS (PRINT (VARIABLES LASTPOS T)))
```

(**BREAKREAD** *TYPE*)   [Function]

Useful within BREAKMACROS for reading arguments. If BRKCOMS is non-NIL (the command in which the call to BREAKREAD appears was not typed in), returns the next break command from BRKCOMS, and sets BRKCOMS to (CDR BRKCOMS).

If BRKCOMS is NIL (the command was typed in), then BREAKREAD returns either the rest of the commands on the line as a list (if *TYPE* = LINE) or just the next command on the line (if *TYPE* is not LINE).

For example, the BT command is defined as (BAKTRACE LASTPOS NIL (BREAKREAD 'LINE) 0 T). Thus, if you type BT, the third argument to BAKTRACE is NIL. If you type BT SUBRP, the third argument is (SUBRP).

BREAKRESETFORMS   [Variable]

If you are developing programs that change the way a user and Medley normally interact (e.g., change or disable the interrupt or line-editing characters, turn off echoing, etc.), debugging them by breaking or tracing may be difficult, because Medley might be in a "funny" state at the time of the break. BREAKRESETFORMS is designed to solve this problem. You put in BREAKRESETFORMS expressions suitable for use in conjunction with RESETFORM or RESETSAVE (see Changing and Restoring System State below). When a break occurs, BREAK1 evaluates each expression on BREAKRESETFORMS *before* any interaction with the terminal, and saves the values. When the break expression is evaluated via an EVAL, OK, or GO, BREAK1 first restores the state of the system with respect to the various expressions on BREAKRESETFORMS. When control returns to BREAK1, the expressions on BREAKRESETFORMS are *again* evaluated, and their values saved. When the break is exited with an OK, GO, RETURN, or ↑ command, by typing Control-D, or by a RETFROM or RETEVAL you type in, BREAK1 again restores state. Thus the net effect is to make the break invisible with respect to your programs, but nevertheless allow you to interact in the break in the normal fashion.

All user type-in is scanned to make the operations undoable, as described in Chapter 13. At this point, RETFROMs and RETEVALs are also noticed. However, if you type in an expression which calls a function that then does a RETFROM, this RETFROM will not be noticed, and the effects of BREAKRESETFORMS will *not* be reversed.

As mentioned earlier, BREAK1 detects "Break within a break" situations, and avoids infinite loops. If the loop occurs because of an error, BREAK1 simply rebinds BREAKRESETFORMS to NIL, and calls HELP. This situation most frequently occurs when there is a bug in a function called by BREAKRESETFORMS.

SETQ expressions can also be included on BREAKRESETFORMS for saving and restoring system parameters, e.g. (SETQ LISPXHISTORY NIL), (SETQ DWIMFLG NIL), etc. These are handled specially by BREAK1 in that the current value of the variable is saved before the SETQ is executed, and upon restoration, the variable is set back to this value.

Signalling Errors

With the Medley release, Interlisp errors use the Xerox Common Lisp (XCL) error system. Most of the functions still exist for compatibility with previous releases, but the underlying machinery has changed. There are some incompatible differences, especially with respect to error numbers. All errors are now handled by signalling an object of type XCL:CONDITION. This means the error numbers generated are different from the old Interlisp method of registered numbers for well-known errors and error messages for all other errors. The mapping from Interlisp erors to Lisp error conditions is listed in the Error List sections below. The obsolete error numbers still generate error messages, but they are useless.

(**ERRORX** *ERXM*)   [Function]

Calls CL:ERROR after first converting *ERXM* into a condition. If *ERXM* is NIL the value of *LAST-CONDITION* is used. If *ERXM* is an Interlisp error descriptor, it is first converted to a condition. If *ERXM* is already a condition, it is passed along unchanged. ERRORX also sets a proceed case for XCL:PROCEED, which will attempt to re-evaluate the caller of ERRORX, much as OK did in older versions of the break package.

(**ERROR** *MESS₁ MESS₂ NOBREAK*)   [Function]

Prints *MESS₁* (using PRIN1), followed by a space if *MESS₁* is an atom, otherwise a carriage return. Then *MESS₂* is printed (using PRIN1 if *MESS₂* is a string; otherwise PRINT). For example, (ERROR "NON-NUMERIC ARG" T) prints

    NON-NUMERIC ARG
    T

and (ERROR 'FOO "NOT A FUNCTION") prints FOO NOT A FUNCTION. If both *MESS₁* and *MESS₂* are NIL, the message printed is simply ERROR.

If *NOBREAK* = T, ERROR prints its message and then calls ERROR! (below). Otherwise it calls (ERRORX '(17 (*MESS₁* . *MESS₂*))), i.e., generates error number 17, in which case the decision as to whether to break, and whether to print a message, is handled as any other error.

If the value of HELPFLAG (see the Controlling When to Break section above) is BREAK!, a break will always occur, irregardless of the value of *NOBREAK*.

If ERROR causes a break, the "break expression" is (ERROR *MESS₁* *MESS₂* *NOBREAK*). Using the GO, OK, , or EVAL break commands (see the Break Commands section above) will simply call ERROR again. It is sometimes helpful to design programs that call ERROR such that if the call to ERROR returns (as the result of using the RETURN break command), the operation is tried again. This lrts you fix any problems within the break environment, and try to continue the operation.

(**HELP** *MESS₁ MESS₂ BRKTYPE*)   [Function]

Prints *MESS₁* and *MESS₂* similar to ERROR, and then calls BREAK1 passing *BRKTYPE* as the BRKTYPE argument. If both *MESS₁* and *MESS₂* are NIL, Help! is used for the message. HELP is a convenient way to program a default condition, or to terminate some portion of a program which the computation is theoretically never supposed to reach.

(**SHOULDNT** *MESS*)   [Function]

> Useful in situations when a program detects a condition that should never occur. Calls HELP with the message arguments *MESS* and "Shouldn't happen!" and a BRKTYPE argument of 'ERRORX.

(**ERROR!**)   [Function]

> Equivalent to XCL:ABORT, except that if no ERRORSET or XCL:CATCH-ABORT isa found, it unwinds to the top of the process.

(**RESET**)   [Function]

> Programmable Control-D; immediately returns to the top level.

LAST-CONDITION   [Variable]

> Value is the condition object most recently signaled.

(**SETERRORN** *NUM MESS*)   [Function]

> Converts its arguments into a condition, then sets the value of *LAST-CONDITION* to the result.

(**ERRORMESS** *U*)   [Function]

> Prints message corresponding to its first argument. For example, (ERRORMESS '(17 T)) would print: T is not a LIST

(**ERRORMESS1** *MESS₁ MESS₂ MESS₃*)   [Function]

> Prints the message corresponding to a HELP or ERROR break.

(**ERRORSTRING** *X*)   [Function]

> Returns as a new string the message corresponding to error number *X*, e.g., (ERRORSTRING 10) = "NON-NUMERIC ARG".

Catching Errors

All error conditions are not caused by program bugs. For some programs, it is reasonable for some errors to occur (such as file not found errors) and it is possible for the program to handle the error itself. There are a number of functions that allow a program to "catch" errors, rather than abort the computation or cause a break.

(**ERRORSET** *FORM FLAG*)   [Function]

> Performs (EVAL *FORM*). If no error occurs in the evaluation of *FORM*, the value of ERRORSET is a list containing one element, the value of (EVAL *FORM*). If an error did occur, the value of ERRORSET is NIL.

> ERRORSET is a lambda function, so its arguments are evaluated *before* it is entered, i.e., (ERRORSET X) means EVAL is called with the *value* of X. In most cases, ERSETQ and NLSETQ (below) are more useful.

> **Note**: Beginning with the Medley release, there are no longer frames named `ERRORSET` on the stack and any programs that explicity look for them must be changed.

> **Performance Note**: When a call to `ERSETQ` or `NLSETQ` is compiled, the form to be evaluated is compiled as a separate function. However, compiling a call to `ERRORSET` does not compile *FORM*. Therefore, if *FORM* performs a lengthy computation, using `ERSETQ` or `NLSETQ` can be much more efficient than using `ERRORSET`.

> The argument *FLAG* controls the printing of error messages if an error occurs. If a *break* occurs below an `ERRORSET`, the message is printed regardless of the value of *FLAG*.

> If *FLAG* = `T`, the error message is printed; if *FLAG* = `NIL`, the error message is not printed (unless `NLSETQGAG` is `NIL`, see below).

> If *FLAG* = `INTERNAL`, this `ERRORSET` is ignored for the purpose of deciding whether or not to break or print a message (see the Controlling When to Break section above). However, the `ERRORSET` is in effect for the purpose of flow of control, i.e., if an error occurs, this `ERRORSET` returns `NIL`.

> If *FLAG* = `NOBREAK`, no break will occur, even if the time criterion for breaking is met (the Controlling When to Break section above). *FLAG* = `NOBREAK` will *not* prevent a break from occurring if the error occurs more than `HELPDEPTH` function calls below the errorset, since `BREAKCHECK` will stop searching before it reaches the `ERRORSET`. To guarantee that no break occurs, you would also either have to reset `HELPDEPTH` or `HELPFLAG`.

(**ERSETQ** *FORM*)   [NLambda Function]

> Evaluates *FORM*, letting a break happen if an error occurs, but 9^ brings you back to the `ERSETQ`. Performs (`ERRORSET` '*FORM* `T`), printing error messages.

(**NLSETQ** *FORM*)   [NLambda Function]

> Evaluates *FORM*, witout breaking, returning `NIL` if an error occurs or a list containing *FORM* if no error occurs. Performs (`ERRORSET` '*FORM* `NIL`), without printing error messages.

NLSETQGAG   [Variable]

> If `NLSETQGAG` is `NIL`, error messages will print, regardless of the *FLAG* argument of `ERRORSET`. `NLSETQGAG` effectively changes all `NLSETQ`s to `ERSETQ`s. `NLSETQGAG` is initially `T`.

Changing and Restoring System State

In Medley, a computation can be interrupted/aborted at any point due to an error, or more forcefully, because a Control-D was typed, causing return to the top level. This situation creates problems for programs that need to perform a computation with the system in a "different state", e.g., different radix, input file, readtable, etc. but want to be able to restore the state when the computation has completed. While program errors and Control-E are "caught" by `ERRORSET`s, Control-D is not. The program could redefine Control-D as a user interrupt (see Chapter 30), check for it, reenable it, and

call RESET or something similar. Thus the system may be left in its changed state as a result of the computation being aborted. The following functions address this problem.

These functions cannot handle the situation where their environment is exited via anything other than a normal return, an error, or a reset. Therefore, a RETEVAL, RETFROM, RESUME, etc., will never be seen.

(**RESETLST** *FORM₁ ... FORMₙ*)   [NLambda NoSpread Function]

> RESETLST evaluates its arguments in order, after setting up an ERRORSET so that any reset operations performed by RESETSAVE (see below) are restored when the forms have been evaluated (or an error occurs, or a Control-D is typed). If no error occurs, the value of RESETLST is the value of *FORMₙ*, otherwise RESETLST generates an error (after performing the necessary restorations).
>
> RESETLST compiles open.

(**RESETSAVE** *X Y*)   [NLambda NoSpread Function]

> RESETSAVE is used within a call to RESETLST to change the system state by calling a function or setting a variable, while specifying how to restore the original system state when the RESETLST is exited (normally, or with an error or Control-D).
>
> If *X* is atomic, resets the top level value of *X* to the value of *Y*. For example, (RESETSAVE LISPXHISTORY EDITHISTORY) resets the value of LISPXHISTORY to the value of EDITHISTORY, and provides for the original value of LISPXHISTORY to be restored when the RESETLST completes operation, (or an error occurs, or a Control-D is typed).
>
> **Note**: If the variable is simply rebound, the RESETSAVE will not affect the most recent binding but will change only the top level value, and therefore probably not have the intended effect.
>
> If *X* is not atomic, it is a form that is evaluated. If *Y* is NIL, *X* must return as its value its "former state", so that the effect of evaluating the form can be reversed, and the system state can be restored, by applying CAR of *X* to the value of *X*. For example, (RESETSAVE (RADIX 8)) performs (RADIX 8), and provides for RADIX to be reset to its original value when the RESETLST completes by applying RADIX to the value returned by (RADIX 8).
>
> In the special case that CAR of *X* is SETQ, the SETQ is transparent for the purposes of RESETSAVE, i.e. you could also have written (RESETSAVE (SETQ X (RADIX 8))), and restoration would be performed by applying RADIX, not SETQ, to the previous value of RADIX.
>
> If *Y* is not NIL, it is evaluated (before *X*), and its *value* is used as the restoring expression. This is useful for functions which do not return their "previous setting". For example,
>
> > [RESETSAVE (SETBRK ...) (LIST 'SETBRK (GETBRK]
>
> will restore the break characters by applying SETBRK to the value returned by (GETBRK), which was computed before the (SETBRK ...) expression was evaluated. The restoration expression is "evaluated" by *applying* its CAR to its CDR. This insures that the "arguments" in the CDR are not evaluated again.

If X is NIL, Y is still treated as a restoration expression. Therefore,

```
(RESETSAVE NIL (LIST 'CLOSEF FILE))
```

will cause FILE to be closed when the RESETLST that the RESETSAVE is under completes (or an error occurs or a Control-D is typed).

RESETSAVE can be called when *not* under a RESETLST. In this case, the restoration is performed at the next RESET, i.e., Control-D or call to RESET. In other words, there is an "implicit" RESETLST at the top-level executive.

RESETSAVE compiles open. Its value is not a "useful" quantity.

(**RESETVAR** *VAR NEWVALUE FORM*)   [NLambda Function]

Simplified form of RESETLST and RESETSAVE for resetting and restoring global variables. Equivalent to (RESETLST (RESETSAVE *VAR NEWVALUE*) *FORM*). For example, (RESETVAR LISPXHISTORY EDITHISTORY (FOO)) resets LISPXHISTORY to the value of EDITHISTORY while evaluating (FOO). RESETVAR compiles open. If no error occurs, its value is the value of *FORM*.

(**RESETVARS** *VARSLST* $E_1 E_2 \ldots E_N$)   [NLambda NoSpread Function]

Similar to PROG, except that the variables in *VARSLST* are global variables. In a deep bound system (like Medley), each variable is "rebound" using RESETSAVE.

In a shallow bound system (like Interlisp-10) RESETVARS and PROG are identical, except that the compiler insures that variables bound in a RESETVARS are declared as SPECVARS (see Chapter 18).

RESETVARS, like GETATOMVAL and SETATOMVAL (see Chapter 2), is provided to permit compatibility (i.e. transportablility) between a shallow bound and deep bound system with respect to conceptually global variables.

Note: Like PROG, RESETVARS returns NIL unless a RETURN statement is executed.

(**RESETFORM** *RESETFORM FORM_1 FORM_2 \ldots FORM_N*)   [NLambda NoSpread Function]

Simplified form of RESETLST and RESETSAVE for resetting a system state when the corresponding function returns as its value the "previous setting." Equivalent to (RESETLST (RESETSAVE *RESETFORM*) *FORM_1 FORM_2 \ldots FORM_N*). For example, (RESETFORM (RADIX 8) (FOO)). RESETFORM compiles open. If no error occurs, it returns the value returned by $FORM_N$.

For some applications, the restoration operation must be different depending on whether the computation completed successfully or was aborted somehow (e.g., by an error or by typing Control-D). To facilitate this, while the restoration operation is being performed, the value of RESETSTATE is bound to NIL, ERROR, RESET, or HARDRESET depending on whether the exit was normal, due to an error, due to a reset (i.e., Control-D), or due to call to HARDRESET (see Chapter 23). As an example of the use of RESETSTATE,

```
(RESETLST
  (RESETSAVE (INFILE X)
```

```
(LIST '[LAMBDA (FL)
        (COND ((EQ RESETSTATE 'RESET)
                (CLOSEF FL)
                DELFILE FL]
                    X))
                    FORMS)
```

will cause X to be closed and deleted only if a Control-D was typed during the execution of *FORMS*.

When specifying complicated restoring expressions, it is often necessary to use the old value of the saving expression. For example, the following expression will set the primary input file (to FL) and execute some forms, but reset the primary input file only if an error or Control-D occurs.

```
(RESETLST
        (SETQ TEM (INPUT FL))
        (RESETSAVE NIL
            (LIST '(LAMBDA (X) (AND RESETSTATE (INPUT X)))
                    TEM))
            FORMS)
```

So that you will not have to explicitly save the old value, the variable OLDVALUE is bound at the time the restoring operation is performed to the value of the saving expression. Using this, the previous example could be recoded as:

```
(RESETLST
        (RESETSAVE (INPUT FL)
            '(AND RESETSTATE (INPUT OLDVALUE)))
            FORMS)
```

As mentioned earlier, restoring is performed by applying CAR of the restoring expression to the CDR, so RESETSTATE and (INPUT OLDVALUE) will not be evaluated by the APPLY. This particular example works because AND is an nlambda function that explicitly evaluates its arguments, so APPLYing AND to (RESETSTATE (INPUT OLDVALUE)) is the same as EVALing (AND RESETSTATE (INPUT OLDVALUE)). PROGN also has this property, so you can use a lambda function as a restoring form by enclosing it within a PROGN.

The function RESETUNDO (see Chapter 13) can be used in conjunction with RESETLST and RESETSAVE to provide a way of specifying that the system be restored to its prior state by *undoing* the side effects of the computations performed under the RESETLST.

Error List

There are currently fifty-plus types of errors in Medley. Some of these errors are implementation dependent, i.e., appear in Medley but may not appear in other Interlisp systems. The error number is set internally by the code that detects the error before it calls the error handling functions, and is used by ERRORMESS for printing error messages.

Most errors will print the offending expression as part of the error message. Error number 18 (Control-B) always causes a break (unless HELPFLAG is NIL). All other errors cause breaks if BREAKCHECK returns T (see Controlling When to Break above).

The foling error messages are arranged numerically with the printed message next to the error number. *X* is the offending expression in each error message. The obsolete error numbers still generate error messags, but they aren't particularly useful. For information on how to use the Common Lisp error conditions in your own programs, see *Common Lisp: the Language* by Steele.

0   Obsolete.

1   Obsolete.

2   **Stack Overflow**

Occurs when computation is too deep, either with respect to number of function calls, or number of variable bindings. Usually because of a non-terminating recursive computation, i.e., a bug. Condition type: STACK-OVERFLOW.

3   **RETURN to nonexistant block: *X***

Call to RETURN when not inside of an interpreted PROG. Condition type: ILLEGAL-RETURN.

4   ***X* is not a LIST**

RPLACA called on a non-list. Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'LIST

5   **Device error: *X***

An error with the local disk drive. Condition type: XCL:SIMPLE-DEVICE-ERROR *message*

6   **Serious condition XCL:ATTEMPT-TO-CHANGE-CONSTANT occured.**

Via SET or SETQ. Condition type: XCL:ATTEMPT-TO-CHANGE-CONSTANT

7   **Attempt to rplac NIL with *X***

Attempt either to RPLACA or to RPLACD NIL with something other than NIL. Condition type: XCL:ATTEMPT-TO-RPLAC-NIL *message*

8   **GO to a nonexistant tag: *X*.**

GO when not inside of a PROG, or GO to nonexistent label. Condition type: ILLEGAL-GO *tag*

9   **File won't open: *X***

From OPENSTREAM (see Chapter 24). Condition type: XCL:FILE-WONT-OPEN *pathname*

10 ***X* is not a NUMBER**

A numeric function e.g., PLUS, TIMES, GREATERP, expected a number and didn't get one. Condition type: XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED TYPE 'CL:NUMBER

11 **Symbol name too long**

Attempted to create a symbol (via PACK, or typing one in, or reading from a file) with too many characters. In Medley, the maximum number of characters in a symbol is 255. Condition type: XCL:SYMBOL-NAME-TOO-LONG

12 **Symbol hash table full**

No room for any more (new) atoms. Condition type: XCL:SYMBOL-HT-FULL

13 **Stream not open: *X***

From an I/O function, e.g., READ, PRINT, CLOSEF. Condition type: XCL:STREAM-NOT-OPEN *stream*

14 ***X* is not a SYMBOL.**

SETQ, PUTPROP, GETTOPVAL, etc., given a non-atomic argument. Condition type: XCL:SMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'CL:SYMBOL

15   Obsolete

16   **End of file *X***

From an input function, e.g., READ, READC, RATOM. After the error occurs, the file will still be left
open. Condition type: END-OF-FILE *stream*

17   ***X varying messages.***

Call to ERROR (see Signalling Errors above). Condition type: INTERLISP-ERROR MESSAGE

18   Obsolete

19   **Illegal stack arg: *X***

A stack function expected a stack position and was given something else. This might occur if the
arguments to a stack function are reversed. Also occurs if you specified a stack position with a
function name, and that function was not found on the stack (see Chapter 11). Condition type:
ILLEGAL-STACK-ARG *arg*.

20   Obsolete

21   **Array space full**

System will first initiate a garbage collection of array space, and if no array space is reclaimed,
will then generate this error. Condition type: XCL:ARRAY-SPACE-FULL.

22   **File system resources exceeded: *X***

Includes no more disk space, disk quota exceeded, directory full, etc. Condition type: XCL:FS-
RESOURCE-EXCEEDED

23   **File not found**

File name does not correspond to a file in the corresponding directory. Can also occur if file name
is ambiguous. Condition type: XCL:FILE-NOT-FOUND *pathname*

24   Obsolete

25   **Invalid argument: *X***

A form ends in a non-list other than NIL, e.g., (CONS T . 3). Condition type: INVALID-
ARGUMENT-LIST *argument*

26   **Hash table full: *X***

See hash array functions, Chapter 6. Condition type: XCL:HASH-TABLE-FULL *table*

27   **Invalid argument: *X***

Catch-all error. Currently used by PUTD, EVALA, ARG, FUNARG, etc. Condition type: INVALID-
ARGUMENT-LIST *argument*

28   **X is not a ARRAYP.**

ELT or SETA given an argument that is not a legal array (see Chapter 5). Condition type:
XCL:SIMPLE-TYPE-ERROR *culprit* :EXPECTED-TYPE 'ARRAYP

29   Obsolete

30   **Stack ptr ahs been released NOBIND**

A released stack pointer was supplied as a stack descriptor for a purpose other than as a stack
pointer to be re-used (see Chapter 11). Condition type: STACK-POINTER-REALEASED *name*

31 **Serious condition XCL:STORAGE-EXHAUSTED occured.**

Following a garbage collection, if not enough words have been collected, and there is no un-allocated space left in the system, this error is generated. Condition type: `XCL:STORAGE-EXHAUSTED`

32 Obsolete

33 Obsolete

34 **No more data types available**

All available user data types have been allocated (see Chapter 8). Condition type: `XCL:DATA-TYPES-EXHAUSTED`

35 **Serious condition XCL:ATTEMPT-TO-CHANGE-CONSTANT occured.**

In a `PROG` or `LAMBDA` expression. Condition type: `XCL:ATTEMPT-TO-CHANGE-CONSTANT`

36 Obsolete

37 Obsolete

38 *X* **is not a READTABLEP.**

The argument was expected to be a valid read table (see Chapter 25). Condition type: `XCL:SIMPLE-TYPE-ERROR` *culprit* `:EXPECTED-TYPE 'READTABLEP`

39 *X* **is not a TERMTABLEP.**

The argument was expected to be a valid terminal table (see Chapter 30). Condition type: `XCL:SIMPLE-TYPE-ERROR` *culprit* `:EXPECTED-TYPE 'TERMTABLEP`

40 Obsolete

41 **Protection violation:** *X*

Attempt to open a file that you do not have access to. Also reference to unassigned device. Condition type: `XCL:FS-PROTECTION-VIOLATION`

42 **Invalid pathname:** *X*

Illegal character in file specification, illegal syntax, e.g. two ;'s etc. Condition type: `XCL:INVALID-PATHNAME` *pathname*

43 Obsolete

44 *X* **is an unbound variable**

This occurs when a variable (symbol) was used which had neither a stack binding (wasn't an argument to a function nor a `PROG` variable) nor a top level value. The "culprit" ((`CADR ERRORMESS`)) is the symbol. If DWIM corrects the error, no error occurs and the error number is not set. However, if an error is going to occur, whether or not it will cause a break, the error number will be set. Condition type: `UNBOUND-VARIABLE` *name*

45 **Serious condition UNDEFINED-CAR-OF-FORM occured.**

Undefined function error. This occurs when a form is evaluated whose function position (`CAR`) does not have a definition as a function. Condition type: `UNDEFINE-CAR-OF FORM` *function*

46   ***X varying messages.***

This error is generated if `APPLY` is given an undefined function. Culprit is *(LIST FN ARGS)*
Condition type: `UNDEFINED-FUNCTION-IN-APPLY`

47   **CONTROL E**

Control-E was typed. Condition type: `XCL:CONTROL-E-INTERRUPT`

48   **Floating point underflow.**

Underflow during floating-point operation. Condition type: `XCL:FLOATING-UNDERFLOW`

49   **Floating point overflow.**

Overflow during floating-point operation. Condition type: `XCL:OVERFLOW`

50   Obsolete

51   ***X is not a HASH-TABLE***

Hash array operations given an argument that is not a hash array. Condition type:
`XCL:SIMPLE-TYPE-ERROR` *culprit* `:EXPECTED-TYPE 'CL:HASH-TABLE`

52   **Too many arguments to *X***

Too many arguments given to a lambda-spread, lambda-nospread, or nlambda-spread function.

Medley does not cause an error if more arguments are passed to a function than it is defined with.
This argument occurs when more individual arguments are passed to a function than Medley can
store on the stack at once. The limit is currently 80 arguments.

In addition, many system functions, e.g., `DEFINE`, `ARGLIST`, `ADVISE`, `LOG`, `EXPT`, etc, also
generate errors with appropriate messages by calling `ERROR` (see Signalling Errors above) which
causes error number 17. Condition type: `TOO-MANY-ARGUMENTS` *callee* `:MAXIMUM`
`CL:CALL-ARGUMENTS-LIMIT`

MEDLEY REFERENCE MANUAL

[This page intentionally left blank]

Medley provides several different facilities for modifing the behavior of a function without actually editing its definition. By "breaking" a function, you can cause breaks to occur at various times in the running of an incomplete program, so that the program state can be inspected. "Tracing" a function causes information to be printed every time the function is entered or exited.

"Advising" is a facility for specifying longer-term function modifications. Even system functions can be changed through advising.

Breaking Functions and Debugging

Debugging a collection of Lisp functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. In the Medley, there are three facilities which allow you to (temporarily) modify selected function definitions so that you can follow the flow of control in your programs, and obtain this debugging information. All three redefine functions in terms of a system function, BREAK1 (see Chapter 14).

BREAK modifies the definition of a function *FN*, so that whenever *FN* is called and a break condition (user-defined) is satisfied, a function break occurs. You can then interrogate the state of the machine, perform any computation, and continue or return from the call.

TRACE modifies a definition of a function *FN* so that whenever *FN* is called, its arguments (or some other user-specified values) are printed. When the value of *FN* is computed it is printed also. TRACE is a special case of BREAK.

BREAKIN allows you to insert a breakpoint inside an expression defining a function. When the breakpoint is reached and if a break condition (defined by you) is satisfied, a temporary halt occurs and you can again investigate the state of the computation.

The following two examples illustrate these facilities. In the first example, the function FACTORIAL is traced. TRACE redefines FACTORIAL so that it print its arguments and value, and then goes on with the computation. When an error occurs on the fifth recursion, a full interactive break occurs. The situation is then the same as though (BREAK FACTORIAL) had been performed instead of (TRACE FACTORIAL), now you can evaluate various Interlisp forms and direct the course of the computation. In this case, the variable N is examined, and BREAK1 is instructed to return 1 as the value of this cell to FACTORIAL. The rest of the tracing proceeds without incident. Presumably, FACTORIAL would be edited to change L to 1.

```
←PP FACTORIAL
(FACTORIAL
        [LAMBDA (N)
        (COND
           ((ZEROP N)
             L)
           (T (ITIMES N (FACTORIAL (SUB1 N]))
FACTORIAL
←(TRACE FACTORIAL)
(FACTORIAL)
←(FACTORIAL 4)
FACTORIAL:
N = 4
```

```
     FACTORIAL:
    N = 3
        FACTORIAL:
        N = 2
            FACTORIAL:
            N = 1
                FACTORIAL:
                N = 0
UNBOUND ATOM
L
(FACTORIAL BROKEN)
:N
0
:RETURN 1
                FACTORIAL = 1
            FACTORIAL = 1
        FACTORIAL = 2
    FACTORIAL = 6
FACTORIAL = 24
24
←
```

In the second example, a non-recursive definition of FACTORIAL has been constructed. BREAKIN is used to insert a call to BREAK1 just after the PROG label LOOP. This break is to occur only on the last two iterations, when N is less than 2. When the break occurs, in trying to look at the value of N, NN is mistakenly typed. The break is maintained, however, and no damage is done. After examining N and M the computation is allowed to continue by typing OK. A second break occurs after the next iteration, this time with N = 0. When this break is released, the function FACTORIAL returns its value of 120.

```
←PP FACTORIAL
(FACTORIAL
  [LAMBDA (N)
      (PROG ((M 1))
        LOOP (COND
                    ((ZEROP N)
                     (RETURN M)))
              (SETQ M (ITIMES M N))
              (SETQ N (SUB1 N))
              (GO LOOP])
FACTORIAL

←(BREAKIN FACTORIAL (AFTER LOOP) (ILESSP N 2]
SEARCHING...
FACTORIAL

←((FACTORIAL 5)
((FACTORIAL) BROKEN)
:NN
U.B.A.
NN
(FACTORIAL BROKEN AFTER LOOP)
:N
1
:M
120
:OK
(FACTORIAL)
```

```
((FACTORIAL) BROKEN)
:N
0
:OK
(FACTORIAL)
120
←
```

Note: BREAK and TRACE can also be used on CLISP words which appear as CAR of form, e.g. FETCH, REPLACE, IF, FOR, DO, etc., even though these are not implemented as functions. For conditional breaking, you can refer to the entire expression via the variable EXP, e.g. (BREAK (FOR (MEMB 'UNTIL EXP))).

(**BREAK0** *FN WHEN COMS* − −)   [Function]

> Sets up a break on the function *FN*; returns *FN*. If *FN* is not defined, returns (*FN* NOT DEFINED).

> The value of *WHEN*, if non-NIL, should be an expression that is evaluated whenever *FN* is entered. If the value of the expression is non-NIL, a break is entered, otherwise the function simply called and returns without causing a break. This provides the means of conditionally breaking a function.

> The value of *COMS*, if non-NIL, should be a list of break commands, that are interpreted and executed if a break occurs. (See the *BRKCOMS* argument to BREAK1, Chapter 14.)

> BREAK0 sets up a break by doing the following:

>> Redefines *FN* as a call to BREAK1 (Chapter 14), passing an equivalent definition of *FN*, *WHEN*, *FN*, and *COMS* as the BRKEXP, *BRKWHEN*, *BRKFN*, and *BRKCOMS* arguments to BREAK1

>> Defines a GENSYM (Chapter 2) with the original definition of *FN*, and puts it on the property list of *FN* under the property BROKEN

>> Puts the form (BREAK0 *WHEN COMS*) on the property list of *FN* under the property BRKINFO (for use in conjunction with REBREAK)

>> Adds *FN* to the front of the list BROKENFNS.

> If *FN* is non-atomic and of the form (*FN$_1$* IN *FN$_2$*), BREAK0 breaks every call to *FN$_1$* from within *FN$_2$*. This is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g., (RPLACA IN FOO), (PRINT IN FIE), etc. It is similar to BREAKIN described below, but can be performed even when *FN$_2$* is compiled or blockcompiled, whereas BREAKIN only works on interpreted functions. If *FN$_1$* is not found in *FN$_2$*, BREAK0 returns the value (*FN$_1$* NOT FOUND IN *FN$_2$*).

> BREAK0 breaks one function *inside* another by first calling a function which changes the name of *FN$_1$* wherever it appears inside of *FN$_2$* to that of a new function, *FN1–IN–FN2*, which is initially given the same function definition as *FN$_1$*. Then BREAK0 proceeds to

break on *FN1*-IN-*FN2* exactly as described above. In addition to breaking *FN1*-IN-*FN2* and adding *FN1*-IN-*FN2* to the list BROKENFNS, BREAK0 adds *FN1* to the property value for the property NAMESCHANGED on the property list of *FN₂* and puts (*FN₂* . *FN₁*) on the property list of *FN1*-IN-*FN2* under the property ALIAS. This will enable UNBREAK to recognize what changes have been made and restore the function *FN₂* to its original state.

If *FN* is nonatomic and not of the above form, BREAK0 is called for each member of *FN* using the same values for *WHEN*, *COMS*, and *FILE*. This distributivity permits you to specify complicated break conditions on several functions. For example,

```
(BREAK0 '(FOO1 ((PRINT PRIN1) IN (FOO2 FOO3)))
    '(NEQ X T)
    '(EVAL ?= (Y Z) OK) )
```

will break on FOO1, PRINT-IN-FOO2, PRINT-IN-FOO3, PRIN1-IN-FOO2 and PRIN1-IN-FOO3.

If *FN* is non-atomic, the value of BREAK0 is a list of the functions broken.

(**BREAK** *X*)   [NLambda NoSpread Function]

For each atomic argument, it performs (BREAK0 *ATOM* T). For each list, it performs (APPLY 'BREAK0 *LIST*). For example, (BREAK FOO1 (FOO2 (GREATERP N 5) (EVAL))) is equivalent to (BREAK0 'FOO1 T) and (BREAK0 'FOO2 '(GREATERP N 5) '(EVAL)).

(**TRACE** *X*)   [NLambda NoSpread Function]

For each atomic argument, it performs (BREAK0 *ATOM* T '(TRACE ?= NIL GO)). The flag TRACE is checked for in BREAK1 and causes the message "*FUNCTION* :" to be printed instead of (*FUNCTION* BROKEN).

For each list argument, CAR is the function to be traced, and CDR the forms to be viewed, i.e., TRACE performs:

```
(BREAK0 (CAR LIST) T (LIST 'TRACE '?= (CDR LIST) 'GO))
```

For example, (TRACE FOO1 (FOO2 Y)) causes both FOO1 and FOO2 to be traced. All the arguments of FOO1 are printed; only the value of Y is printed for FOO2. In the special case when you want to see *only* the value, you can perform (TRACE (*FUNCTION*)). This sets up a break with commands (TRACE ?= (NIL) GO).

Note: You can always call BREAK0 to obtain combination of options of BREAK1 not directly available with BREAK and TRACE. These two functions merely provide convenient ways of calling BREAK0, and will serve for most uses.

Note: BREAK0, BREAK, and TRACE print a warning if you try to modify a function on the list UNSAFE.TO.MODIFY.FNS (Chapter 10).

(**BREAKIN** *FN WHERE WHEN COMS*)   [NLambda Function]

BREAKIN enables you to insert a break, i.e., a call to BREAK1 (Chapter 14), at a specified location in the interpreted function *FN*. BREAKIN can be used to insert breaks before or after PROG labels, particular SETQ expressions, or even the evaluation of a variable. This

is because `BREAKIN` operates by calling the editor and actually inserting a call to `BREAK1` at a specified point *inside* of the function. If *FN* is a compiled function, `BREAKIN` returns (*FN* `UNBREAKABLE`) as its value.

WHEN should be an expression that is evaluated whenever the break is entered. If the value of the expression is non-`NIL`, a break is entered, otherwise the function simply called and returns without causing a break. This provides the means of creating a conditional break. For `BREAKIN`, unlike `BREAK0`, if *WHEN* is `NIL`, it defaults to `T`.

COMS, if non-`NIL`, should be a list of break commands, that are interpreted and executed if a break occurs. (See the *BRKCONMS* argument to `BREAK1`, Chapter 14.)

WHERE specifies where in the definition of *FN* the call to `BREAK1` is to be inserted. *WHERE* should be a list of the form (`BEFORE ...`), (`AFTER ...`), or (`AROUND ...`). You specify where the break is to be inserted by a sequence of editor commands, preceded by one of the symbols `BEFORE`, `AFTER`, or `AROUND`, which `BREAKIN` uses to determine what to do once the editor has found the specified point, i.e., put the call to `BREAK1` `BEFORE` that point, `AFTER` that point, or `AROUND` that point. For example, (`BEFORE COND`) will insert a break before the first occurrence of `COND`, (`AFTER COND 2 1`) will insert a break after the predicate in the first `COND` clause, (`AFTER BF (SETQ X &)`) after the *last* place X is set. Note that (`BEFORE TTY:`) or (`AFTER TTY:`) permit you to type in commands to the editor, locate the correct point, and verify it, and exit from the editor with `OK`. `BREAKIN` then inserts the break `BEFORE`, `AFTER`, or `AROUND` that point.

Note: A `STOP` command typed to `TTY:` produces the same effect as an unsuccessful edit command in the original specification, e.g., (`BEFORE CONDD`). In both cases, the editor aborts, and `BREAKIN` types (`NOT FOUND`).

If *WHERE* is (`BEFORE ...`) or (`AFTER ...`), the break expression is `NIL`, since the value of the break is irrelevant. For (`AROUND ...`), the break expression will be the indicated form. In this case, you can use the `EVAL` command to evaluate that form, and examine its value, before allowing the computation to proceed. For example, if you inserted a break after a `COND` predicate, e.g., (`AFTER (EQUAL X Y)`), you would be powerless to alter the flow of computation if the predicate were not true, since the break would not be reached. However, by breaking (`AROUND (EQUAL X Y)`), you can evaluate the break expression, i.e., (`EQUAL X Y`), look at its value, and return something else if desired.

If *FN* is interpreted, `BREAKIN` types `SEARCHING...` while it calls the editor. If the location specified by *WHERE* is not found, `BREAKIN` types (`NOT FOUND`) and exits. If it is found, `BREAKIN` puts `T` under the property `BROKEN-IN` and (*WHERE WHEN COMS*) under the the property `BRKINFO` on the property list of *FN*, and adds *FN* to the front of the list `BROKENFNS`.

Multiple break points, can be inserted with a single call to `BREAKIN` by using a list of the form ((`BEFORE ...`) `...` (`AROUND ...`)) for *WHERE*. It is also possible to call `BREAK` or `TRACE` on a function which has been modified by `BREAKIN`, and conversely to `BREAKIN` a function which has been redefined by a call to `BREAK` or `TRACE`.

The message typed for a BREAKIN break is ((*FN*) BROKEN), where *FN* is the name of the function inside of which the break was inserted. Any error, or typing control-E, will cause the full identifying message to be printed, e.g., (FOO BROKEN AFTER COND 2 1).

A special check is made to avoid inserting a break inside of an expression headed by any member of the list NOBREAKS, initialized to (GO QUOTE *), since this break would never be activated. For example, if (GO L) appears before the label L, BREAKIN (AFTER L) will not insert the break inside of the GO expression, but skip this occurrence of L and go on to the next L, in this case the label L. Similarly, for BEFORE or AFTER breaks, BREAKIN checks to make sure that the break is being inserted at a "safe" place. For example, if you request a break (AFTER X) in (PROG ... (SETQ X &) ...), the break will actually be inserted after (SETQ X &), and a message printed to this effect, e.g., BREAK INSERTED AFTER (SETQ X &).

(**UNBREAK** *X*)   [NLambda NoSpread Function]

UNBREAK takes an indefinite number of functions modified by BREAK, TRACE, or BREAKIN and restores them to their original state by calling UNBREAK0. Returns list of values of UNBREAK0.

(UNBREAK) will unbreak all functions on BROKENFNS, in reverse order. It first sets BRKINFOLST to NIL.

(UNBREAK T) unbreaks just the first function on BROKENFNS, i.e., the most recently broken function.

(**UNBREAK0** *FN* —)   [Function]

Restores *FN* to its original state. If *FN* was not broken, value is (NOT BROKEN) and no changes are made. If *FN* was modified by BREAKIN, UNBREAKIN is called to edit it back to its original state. If *FN* was created from (*FN$_1$* IN *FN$_2$*), (i.e., if it has a property ALIAS), the function in which *FN* appears is restored to its original state. All dummy functions that were created by the break are eliminated. Adds property value of BRKINFO to the front of BRKINFOLST.

Note: (UNBREAK0 '(*FN$_1$* IN *FN$_2$*)) is allowed: UNBREAK0 will operate on (*FN$_1$*-IN-*FN$_2$*) instead.

(**UNBREAKIN** *FN*)   [Function]

Performs the appropriate editing operations to eliminate all changes made by BREAKIN. *FN* may be either the name or definition of a function. Value is *FN*.

UNBREAKIN is automatically called by UNBREAK if *FN* has property BROKEN-IN with value T on its property list.

(**REBREAK** *X*)   [NLambda NoSpread Function]

Nlambda nospread function for rebreaking functions that were previously broken without having to respecify the break information. For each function on *X*, REBREAK searches BRKINFOLST for break(s) and performs the corresponding operation. Value is a

list of values corresponding to calls to BREAK0 or BREAKIN. If no information is found for a particular function, returns (*FN*-NO BREAK INFORMATION SAVED).

(REBREAK) rebreaks everything on BRKINFOLST, so (REBREAK) is the inverse of (UNBREAK).

(REBREAK T) rebreaks just the first break on BRKINFOLST, i.e., the function most recently unbroken.

(**CHANGENAME** *FN FROM TO*) [Function]

Replaces all occurrences of *FROM* by *TO* in the definition of *FN*. If *FN* is defined by an expr definition, CHANGENAME performs (ESUBST *TO FROM* (GETD *FN*)) (see Chapter 16). If *FN* is compiled, CHANGENAME searches the literals of *FN* (and all of its compiler generated subfunctions), replacing each occurrence of *FROM* with *TO*.

Note that *FROM* and *TO* do not have to be functions, e.g., they can be names of variables, or any other literals.

CHANGENAME returns *FN* if at least one instance of *FROM* was found, otherwise NIL.

(**VIRGINFN** *FN FLG*) [Function]

The function that knows how to restore functions to their original state regardless of any amount of breaks, breakins, advising, compiling and saving exprs, etc. It is used by PRETTYPRINT, DEFINE, and the compiler.

If *FLG* = NIL, as for PRETTYPRINT, it does not modify the definition of *FN* in the process of producing a "clean" version of the definition; it works on a copy.

If *FLG* = T, as for the compiler and DEFINE, it physically restores the function to its original state, and prints the changes it is making, e.g., FOO UNBROKEN, FOO UNADVISED, FOO NAMES RESTORED, etc.

Returns the virgin function definition.

Advising

The operation of advising gives you a way of modifying a function without necessarily knowing how the function works or even what it does. Advising consists of modifying the *interface* between functions as opposed to modifying the function definition itself, as in editing. BREAK, TRACE, and BREAKDOWN, are examples of the use of this technique: they each modify user functions by placing relevant computations *between* the function and the rest of the programming environment.

The principal advantage of advising, aside from its convenience, is that it allows you to treat anyone's functions as "black boxes," and to modify them without concern for their contents or details of operations. For example, you could modify SYSOUT to set SYSDATE to the time and date of creation by (ADVISE 'SYSOUT '(SETQ SYSDATE (DATE))).

As with BREAK, advising works equally well on compiled and interpreted functions. Similarly, it is possible to make a change which only operates when a function is called from some other specified function. For example, you can modify the interface between two particular functions, instead of the

interface between one function and the rest of the world. This latter feature is especially useful for changing the *internal* workings of a system function.

For example, suppose you wanted TIME (Chapter 22) to print the results of your measurements to the file FOO instead of the terminal. You can accomplish this by (ADVISE '((PRIN1 PRINT SPACES) IN TIME) 'BEFORE '(SETQQ U FOO)).

Advising PRIN1, PRINT, or SPACES directly would have affected all calls to these frequently used functions, whereas advising ((PRIN1 PRINT SPACES) IN TIME) affects just those calls to PRIN1, PRINT, and SPACES from TIME.

Advice can also be specified to operate after a function has been evaluated. The value of the body of the original function can be obtained from the variable !VALUE, as with BREAK1.

Implementation of Advising

After a function has been modified several times by ADVISE, it will look like:

```
(LAMBDA arguments
    (PROG (!VALUE)
        (SETQ !VALUE
            (PROG NIL
                advice1
                    .
                    .          advice before
                    .
                advicen
                (RETURN BODY)))
        advice1
            .
            .      advice after
            .
        advicem
        (RETURN !VALUE)))
```

where *BODY* is equivalent to the original definition. If *FN* was originally an expr definition, *BODY* is the body of the definition, otherwise a form using a GENSYM which is defined with the original definition.

The structure of a function modified by ADVISE allows a piece of advice to bypass the original definition by using the function RETURN. For example, if (COND ((ATOM X) (RETURN Y))) were one of the pieces of advice *before* a function, and this function was entered with X atomic, Y would be returned as the value of the inner PROG, !VALUE would be set to Y, and control passed to the advice, if any, to be executed AFTER the function. If this same piece of advice appeared *after* the function, Y would be returned as the value of the entire advised function.

The advice (COND ((ATOM X) (SETQ !VALUE Y))) *after* the function would have a similar effect, but the rest of the advice *after* the function would still be executed.

Note: Actually, ADVISE uses its own versions of PROG, SETQ, and RETURN, (called ADV-PROG, ADV-SETQ, and ADV-RETURN) to enable advising these functions.

Advise Functions

ADVISE is a function of four arguments: *FN*, *WHEN*, *WHERE*, and *WHAT*. *FN* is the function to be modified by advising, *WHAT* is the modification, or piece of advice. *WHEN* is either BEFORE, AFTER, or AROUND, and indicates whether the advice is to operate BEFORE, AFTER, or AROUND the body of the function definition. *WHERE* specifies exactly where in the list of advice the new advice is to be placed, e.g., FIRST, or (BEFORE PRINT) meaning before the advice containing PRINT, or (AFTER 3) meaning after the third piece of advice, or even (: TTY:). If *WHERE* is specified, ADVISE first checks to see if it is one of LAST, BOTTOM, END, FIRST, or TOP, and operates accordingly. Otherwise, it constructs an appropriate edit command and calls the editor to insert the advice at the corresponding location.

Both *WHEN* and *WHERE* are optional arguments, in the sense that they can be omitted in the call to ADVISE. In other words, ADVISE can be thought of as a function of two arguments (ADVISE *FN WHAT*), or a function of three arguments: (ADVISE *FN WHEN WHAT*), or a function of four arguments: (ADVISE *FN WHEN WHERE WHAT*). Note that the advice is always the *last* argument. If *WHEN* = NIL, BEFORE is used. If *WHERE* = NIL, LAST is used.

> (**ADVISE** *FN WHEN WHERE WHAT*) [Function]
>
> > *FN* is the function to be advised, *WHEN* = BEFORE, AFTER, or AROUND, *WHERE* specifies where in the advice list the advice is to be inserted, and *WHAT* is the piece of advice.
> >
> > If *FN* is of the form (*FN$_1$* IN *FN$_2$*), *FN$_1$* is changed to *FN1*-IN-*FN2* throughout *FN$_2$*, as with break, and then *FN1*-IN-*FN2* is used in place of *FN*. If *FN$_1$* and/or *FN$_2$* are lists, they are distributed as with BREAK0.
> >
> > If *FN* is broken, it is unbroken before advising.
> >
> > If *FN* is not defined, an error is generated, NOT A FUNCTION.
> >
> > If *FN* is being advised for the first time, i.e., if (GETP *FN* 'ADVISED) = NIL, a GENSYM is generated and stored on the property list of *FN* under the property ADVISED, and the GENSYM is defined with the original definition of *FN*. An appropriate expr definition is then created for *FN*, using private versions of PROG, SETQ, and RETURN, so that these functions can also be advised. Finally, *FN* is added to the (front of) ADVISEDFNS, so that (UNADVISE T) always unadvises the last function advised.
> >
> > If *FN* has been advised before, it is moved to the front of ADVISEDFNS.
> >
> > If *WHEN* = BEFORE or AFTER, the advice is inserted in *FN*'s definition either BEFORE or AFTER the original body of the function. Within that context, its position is determined by *WHERE*. If *WHERE* = LAST, BOTTOM, END, or NIL, the advice is added following all other advice, if any. If *WHERE* = FIRST or TOP, the advice is inserted as the first piece of advice. Otherwise, *WHERE* is treated as a command for the editor, similar to BREAKIN, e.g., (BEFORE 3), (AFTER PRINT).
> >
> > If *WHEN* = AROUND, the body is substituted for * in the advice, and the result becomes the new body, e.g., (ADVISE 'FOO 'AROUND '(RESETFORM (OUTPUT T) *)). Note that if several pieces of AROUND advice are specified, earlier ones will be embedded inside later ones. The value of *WHERE* is ignored.

Finally (LIST *WHEN WHERE WHAT*) is added (by ADDPROP) to the value of property ADVICE on the property list of *FN*, so that a record of all the changes is available for subsequent use in readvising. Note that this property value is a list of the advice in order of calls to ADVISE, not necessarily in order of appearance of the advice in the definition of *FN*.

The value of ADVISE is *FN*.

If *FN* is non-atomic, every function in *FN* is advised with the same values (but copies) for *WHEN*, *WHERE*, and *WHAT*. In this case, ADVISE returns a list of individual functions.

Note: Advised functions can be broken. However if a function is broken at the time it is advised, it is first unbroken. Similarly, advised functions can be edited, including their advice. UNADVISE will still restore the function to its unadvised state, but any changes to the body of the definition will survive. Since the advice stored on the property list is the same structure as the advice inserted in the function, editing of advice can be performed on either the function's definition or its property list.

(**UNADVISE** *X*)   [NLambda NoSpread Function]

An nlambda nospread like UNBREAK. It takes an indefinite number of functions and restores them to their original unadvised state, including removing the properties added by ADVISE. UNADVISE saves on the list ADVINFOLST enough information to allow restoring a function to its advised state using READVISE. ADVINFOLST and READVISE thus correspond to BRKINFOLST and REBREAK. If a function contains the property READVICE, UNADVISE moves the current value of the property ADVICE to READVICE.

(UNADVISE) unadvises all functions on ADVISEDFNS in reverse order, so that the most recently advised function is unadvised last. It first sets ADVINFOLST to NIL.

(UNADVISE T) unadvises the first function of ADVISEDFNS, i.e., the most recently advised function.

(**READVISE** *X*)   [NLambda NoSpread Function]

An nlambda nospread like REBREAK for restoring a function to its advised state without having to specify all the advise information. For each function on *X*, READVISE retrieves the advise information either from the property READVICE for that function, or from ADVINFOLST, and performs the corresponding advise operation(s). It also stores this information on the property READVICE if not already there. If no information is found for a particular function, value is (*FN* - NO ADVICE SAVED).

(READVISE) readvises everything on ADVINFOLST.

(READVISE T) readvises the first function on ADVINFOLST, i.e., the function most recently unadvised.

A difference between ADVISE, UNADVISE, and READVISE versus BREAK, UNBREAK, and REBREAK, is that if a function is not rebroken between successive (UNBREAK)s, its break information is forgotten. However, once READVISE is called on a function, that function's advice is permanently saved on its property list (under READVICE); subsequent calls to

UNADVISE will not remove it. In fact, calls to UNADVISE update the property READVICE with the current value of the property ADVICE, so that the sequence READVISE, ADVISE, UNADVISE causes the augmented advice to become permanent. The sequence READVISE, ADVISE, READVISE removes the "intermediate advice" by restoring the function to its earlier state.

(**ADVISEDUMP** *X FLG*) [Function]

Used by PRETTYDEF when given a command of the form (ADVISE ...) or (ADVICE ...). If *FLG* = T, ADVISEDUMP writes both a DEFLIST and a READVISE; this corresponds to (ADVISE ...). If *FLG* = NIL, only the DEFLIST is written; this corresponds to (ADVICE ...). In either case, ADVISEDUMP copies the advise information to the property READVICE, thereby making it "permanent" as described above.

16. SEdit - The Structure Editor

Medley's code editors are "structure" editors—they know how to take advantage of Lisp code being represented as lists. One is a display editor named SEdit and the other is a TTY-based editor.

Starting the Editor

The editor is normally called using the following functions:

(DF *FN*)   [NLambda NoSpread Function]

Edit the definition of the function *FN*. DF handles exceptional cases (the function is broken or advised, the definition is on the property list, the function needs to be loaded from a file, etc.) the same as EDITF (see below).

If you call DF with a name that has no function definition, you are prompted with a choise of definers to use.

(DV *VAR*)   [NLambda NoSpread Function]

Edit the value of the variable *VAR*.

(DP *NAME PROP*)   [NLambda NoSpread Function]

Edit property *PROP* of the symbol *NAME*. If *PROP* is not given, the whole property list of *NAME* is edited.

(DC *FILE*)   [NLambda NoSpread Function]

Edit the file package commands (or "filecoms," see Chapter 17) for the file *FILE*.

(ED *NAME OPTIONS*)   [Function]

This function finds out what kind of definiton *NAME* has and lets you edit it.   If NAME has more than one definition (e.g., it's both a function and a macro), you will be prompted for the right one. If NAME has no definition, you'll be asked what kind of definition to create.

Choosing Your Editor

The default editor may be set with EDITMODE:

```
(EDITMODE NEWMODE)                                                    [Function]
```

> If *NEWMODE* is `DISPLAY`, sets the default editor to be SEdit; or the teletype editor (if *NEWMODE* is `TELETYPE`). Returns the previous setting. If *NEWMODE* is `NIL`, returns the previous setting without setting a new editor.

SEdit - The Structure Editor

SEdit is a structure editor. You use a structure editor when you want to edit objects instead of text. SEdit is a part of the environment and operates directly on objects in the system you are running. SEdit behaves differently depending on the type of objects you are editing.

Common Lisp definitions:	SEdit always edits a copy of a Common Lisp definition. The changes made while you edit a function will not be installed until the edit session is complete.
	For example, when you edit a Common Lisp function, you edit the definition of the function and not the executable version of the function. When you end the session the comments will be stripped of the definition and the definition will be installed as the executable version of the function.
Interlisp functions and macros:	SEdit edits the actual structure that will be run, except editing the source for a compiled function. In this case, changes are made and the function is unsaved when you complete the edit session.
All other structures:	Variables, property lists and other structures are edited directly in place, i.e. SEdit installs all changes as they are made.

If you make a severe editing error, you can abort the edit session with an Abort command (see Command Keys, below). This command undoes all changes from the beginning of the edit session and exits from SEdit without changing your environment.

If you change the definition of an object that is being edited in an SEdit window, Medley will ask you if you want to throw away the changes made there.

SEdit supports the standard Copy-Select mechanism in Medley.

An SEdit Session

Whenever you call SEdit, a new SEdit window is created. This SEdit window has its own process. You can make edits in the window, shrink it while you do something else, expand it and edit some more, and finally close the window when you are done.

Throughout an edit session, SEdit remembers everything that you do IN a change history. You can undo and redo edits sequentially. When you end the edit session, SEdit forgets this information and installs the changes in the system.

You signal the end of the session in the following ways:

- Close the window.

- Shrink the window. If you expand the window again, you can continue editing.

- Issue a Completion Command, see below.

SEdit Carets

There are two carets in SEdit, the edit caret and the structure caret. The edit caret appears when characters are edited within a single symbol, string, or comment. Anything you type will appear at the edit caret as part of the item it's in. The edit caret looks like this:

(a b)

The structure caret appears when the edit point is between symbols (or strings or comments), so that anything you type will go into a new one. It looks like this:

(a b)

SEdit changes the caret frequently, depending on where the caret is positioned. The left mouse button positions the edit caret. The middle mouse button positions the structure caret.

The Mouse

The left mouse button selects parts of Lisp structures. The middle mouse button selects whole Lisp structures.

For example; select the Q in LEQ below by pressing the left mouse button when the pointer is over the Q.

(LEQ n 1)

Any characters you type in now will be appended to the symbol `LEQ`.

Selecting the same letter with the middle mouse button selects the whole symbol (this matches TEdit's character/word selection convention), and sets a structure caret between the `LEQ` and the `n`:

(LEQ▲n 1)

Any characters you type in now will form a new symbol between the `LEQ` and the `n`.

Larger structures can be selected in two ways. Use the middle mouse button to position the mouse cursor on the parenthesis of the list you want to edit. Press the mouse button multiple times, without moving the mouse, extends the selection. In the previous example, if the middle button was pressed twice, the list `(LEQ ...)` would be selected:

(LEQ n 1)

Press the button a third time and you will select the list containing the `(LEQ n 1)` to be selected.

The right mouse button positions the mouse cursor for selecting sequences of structures or substructures. Extended selections are indicated by a box enclosing the structures selected. The selection extends in the same mode as the original selection. That is, if the original selection was a character selection, the right button will be used to select more characters in the same atom. Extended selections also have the property of being marked for pending deletion. That is, the selection takes the place of the caret, and anything typed in is inserted in place of the selection.

For example, selecting the `E` by pressing the left mouse button and selecting the `Q` by pressing the right mouse button will produce:

(L EQ n 1)

Similarly, pressing the middle mouse button and then selecting with the right mouse button extends the selection by whole structures. In our example, pressing the middle mouse button to select `LEQ` and pressing the right mouse button to select the `1` will produce:

(LEQ n 1)

This is not the same as selecting the entire list, as above.   Instead, the elements in the list are collectively selected, but the list itself is not.

Gaps

SEdit requires that everything edited must have an underlying Lisp structure at all times. Some characters, such as single quote " ' " have no meaning by themselves, but must be followed by something more. Whenyou type such a character, SEdit puts a "gap" where the rest of the input should go. When you type, the gap is automatically replaced.

A gap looks like: –✕–

After you type a quote, the gap looks like this: ' ⊠ with the gap marked for pending deletion.

Broken Atoms

When you type an atom (a symbol or a number), SEdit saves the characters you type until you are finished. Typing any character that cannot belong to an atom, like a space or open parenthesis, ends the atom. SEdit then tries to create an atom with the characters you just typed, just as if they were read by the Lisp reader. The atom then becomes part of the structure you're editing.

If an error occurs when SEdit reads the atom, SEdit creates a structure called a Broken-Atom. A Broken-Atom looks and behaves just like a normal atom, but is printed in italics to tell you that something is wrong.

SEdit creates a Broken-Atom when the characters typed don't make a legal atom. For example, the characters "DECLARE:" can't be a symbol because the colon is a package specifier, but the form is not correct for a package-qualified symbol. Similarly, the characters "#b123" cannot represent an integer in base two, because 2 and 3 aren't legal digits in base two, so SEdit would make a Broken-Atom that looks like *#b123*.

You can edit Broken-Atoms just like real atoms. Whenever you finish editing a Broken-Atom, SEdit again tries to create an atom from the characters. If SEdit succeeds, it reprints the atom in SEdit's default font, rather than in italics. Be sure to correct any Broken-Atoms you create before exiting SEdit, since Broken-Atoms do not behave in any useful way outside SEdit.

Special Characters

Some characters have special meanings in Lisp, and are therefor treated specially by SEdit. SEdit must always have a complete structure to work on at any level of the edit. This means that SEdit needs a special way to type in structures such as lists, strings, and quoted objects. In most instances these structures can be typed in just as they would be to a regular Exec, but in the following cases this is not possible.

Lists:	**( )**	Lists begin with an open parenthesis character "(". Typing an open parenthesis gives a balanced list. SEdit inserts both an open and a close parenthesis. The structure caret is placed between the two parentheses. List elements can be typed in at the structure caret. When a close parenthesis, ")" is typed, the caret will be moved outside the list, effectively finishing the list. Square bracket characters, "[" and "]", have no special meaning in SEdit, as they have no special meaning in Common Lisp.
Single Quote:	**'**	
Backquote:	**`**	
Comma:	**,**	
At Sign:	**,@**	
Dot:	**,.**	
Hash Quote:	**#'**	All these characters are special macro characters in Common Lisp. When you type one, SEdit will echo the character followed by a gap, which you should then fill in.
Dotted Lists:	**(.)**	Use period to enter dotted pairs. After you type a dot, SEdit prints a dot and a gap to fill in for the tail of the list. To dot an existing list, point the cursor between the last and second to last elements, and type a dot. To undot a list, select the tail of the list before the dot while holding down the SHIFT key.
Single escape:	**\ or %**	Use the single escape characters to make symbols with special characters. The single escape character for Interlisp is "%". The single escape character for Common Lisp is "\". When you want to create a symbol with a special character in it you have to type a single escape character before you type the character itself. SEdit does not echo the single escape character until you type the following character. For example; create the Common Lisp symbol APAREN-(. Since SEdit normally will treat the "(" as the start of a new list you have to tell SEdit to treat it as an ordinary character. You do this by typing a "\" before you type the "(".
Multiple Escape:	**|**	Use the multiple escape character when you enter symbols with many special characters. SEdit always balances multiple escape characters. When you type one, SEdit adds another, with the caret between them. If you type a second vertical bar, the caret moves after it, but is still inthe same symbol, so you can add more unescaped characters.

Comment:	**;**	A semicolon starts a comment. When you type a semicolon, an empty comment is inserted with the caret in position to type the comment. Comments can be edited like strings.
		There are three levels of comments supported by SEdit: single-, double-, and triple-semicolon. Single-semicolon comments are formatted at the comment column, about three-quarters of the way across the window. Double-semicolon comments are formatted at the current indentation of the code they are in. Triple semicolon comments are formatted against the left margin. The level of a comment can be increased or decreased by pointing after the semicolon, and either typing another semicolon, or backspacing over the preceding semicolon. Comments can be placed anywhere in your Common Lisp code. However, in Interlisp code, they must follow the placement rules for Interlisp comments.
String:	**"**	Enter strings in SEdit by typing a double quote. SEdit balances the double quotes: When one is typed, SEdit produces a second, with the caret between the two. If you type a double-quote in the middle of a string, SEdit breaks the string in two, leaving the caret between them.

SEdit Commands

Enter SEdit commands either from the keyboard or from the SEdit menu. When possible, SEdit uses a named key on the keyboard, e.g., the DELETE key. Other commands are combinations of Meta, Control, and alphabetic keys. For the alphabetic command keys, either uppercase or lowercase will work.

There are two menus available, as an alternative means of invoking commands. They are the middle button popup menu, and the attached command menu. These menus are described in more detail below.

Meta-A	Abort the session. Throw away the changes made to the form.
Meta-B	Change the Print Base. Prompts for entry of the desired Print Base, in decimal. SEdit redisplays fixed point numbers in this new base.
Control-C	Tell SEdit that this session is complete and compiles the definition being edited. The variable `*COMPILE-FN*` determines which function to use as compiler. See the Options section below.
Control-Meta-C	Signals the system that this edit is complete, compiles the definition being editing, and closes the window.
DELETE	Deletes the current selection.

Meta-E   Evaluate the current selection. If the result is a structure, the inspector is called on it, allowing the user to choose how to look at the result. Otherwise, the result is printed in the SEdit prompt window. The evaluation is done in the process from which the edit session was started. Thus, while editing a function from a break window, evaluations are done in the context of the break.

FIND
Meta-F   Find a specified structure, or sequence of structures. If there is a current selection, SEdit looks for the next occurrence of the selected structure. If there is no selection, SEdit prompts for the structure to find, and searches forward from the position of the caret. The found structure will be selected, so the Find command can be used to easily find the same structure again.

If a sequence of structures are selected, SEdit will look for the next occurrence of the same sequence. Similarly, when SEdit prompts for the structure to find, you can type a sequence of structures to look for.

The variable `*WRAP-SEARCH*` controls whether or not SEdit wraps around from the end of the structure being edited and continues searching from the beginning.

Control-Meta-F   Find a specified structure, searching in reverse from the position of the caret.

HELP
Meta-H   Show the argument list for the function currently selected, or currently being typed in, in the SEdit prompt window. If the argument list will not fit in the SEdit prompt window, it is displayed in the main Prompt Window.

Meta-I   Inspect the current selection.

Meta-J   Join any number of sequential Lisp objects of the same type into a single object of that type.   Join is supported for atoms, strings, lists, and comments. In addition, SEdit permits joining of a sequence of atoms and strings, since either type can easily be coerced into the other. In this case, the result of the Join will be an atom if the first object in the selection is an atom, otherwise the result will be a string.

Control-L   Redisplay the structure being edited.

SKIP-NEXT
Meta-N   Select next gap in the structure.

Meta-O   Edit the definition of the current selection. If the selected name has more than one type of definition, SEdit asks for the type to edit. If the selection has no definition, a menu pops up. This menu lets you specify the type of definition to create.

Control-Meta-O   Perform a fast edit by calling `ED` with the `CURRENT` option.

Meta-P   Change the current package for this edit. Prompt the user for a new package name. SEdit will redisplay atoms with respect to that package.

AGAIN
Meta-R   Redo the edit change that was just undone. Redo only works directly following an Undo.   Any number of Undo commands can be sequentially redone.

SHIFT-FIND

Meta-S	Substitute a structure, or sequence of structures within the current selection. SEdit prompts you in the SEdit prompt window for the structures to replace, and the structures to replace with. The selection to substitute within must be a structure selection.
Control-Meta-S	Remove all occurences of a structure or sequence of structures within the current selection. SEdit prompts you for the structures to delete.
UNDO **Meta-U**	Undo the last edit. All changes in the the edit session are remembered, and can be undone sequentially**.**
Control-W	Delete the previous atom or structure. If the caret is in the middle of an atom, deletes backward to the beginning of the atom only.
Control-X	Tell SEdit that this session is complete. The SEdit window remains open.
EXPAND **Meta-X**	Replaces the current selection with its definition. This command can be used to expand macros and translate CLISP.
Control-Meta-X	Tell SEdit that this session is complete Close the SEdit window.
Meta-Z	Mutate. Prompt for a function and call this function with the current selection as the argument. The result is inserted into SEdit and made the current selection. For example, you can replace a structure with its value by selecting it and mutating by EVAL.
Meta-;	Convert old style comments in the selected structure to new style comments. The converter notices any list that begins with the symbol IL:* as an old style comment. Section 16.1.18, Options, describes the converter options .
Control-Meta-;	Put the contents of a structure selection into a comment. This provides an easy way to "comment out" a chunk of code. The Extract command can be used to reverse this process, returning the comment to the structures contained therein.
Meta-/	Extract one level of structure from the current selection. If there is no selection, but there is a structure caret, the list containing the caret is used. This command can be used to strip the parentheses off a list, or to unquote a quoted structure, or to replace a comment with the contained structures.
Meta-' **Meta-`** **Meta-,** **Meta-.** **Meta-@** or **Meta-2** **Meta-#** or **Meta-3** **Meta-.**	Quote the current selection with the specified kind of quote.
Meta-Space **Meta-Return**	Scroll the current selection to the center of the window. Similarly, the Space or Return key can be used to normalize the caret.
Meta-) **Meta-0**	Parenthesize the current selection, position the caret after the new list.

Meta-(
Meta-9   ParenthesizE the current selection, position the caret at the beginning of the new list.

Meta-M   Attach a menu of common commands to the top of the SEdit window. Each SEdit window can have its own menu.

SEdit Command Mnemonics

Abort	Meta-A
Change Print Base	Meta-B
Complete	Control-X
Compile & Complete	Control-C
Close, Compile & Complete	Control-Meta-C
Convert Comment	Meta-;
Make Selection Comment	Control-Meta-;
Previous Delete	Control-W
Selection Delete	DELETE
Selection Dot Comma	Meta-.
Selection At Comma	Meta-@
Edit	Meta-O
Fast Edit	Control-Meta-O
Selection Eval	Meta-E
Macro Expand	Meta-X
Forward Find	Meta-F
Reverse Find	Control-Meta-F
Next Gap	Meta-N
Arglist Help	Meta-H
Inspect	Meta-I
Join	Meta-J
Attach Menu	Meta-M
Expression Mutate	Meat-Z
Change Package	Meta-P
Selection Left Parenthesize	Meta-(
Selection Right Parenthesize	Meta-)
Selection Pop	Meta-/
Selection Back Quote	Meta-'
Selection Hash Quote	Meta-#
Selection Quote	Meta-'
Redisplay	Control-L
Redo	Meta-R
Remove	Control-Meta-S
Substitute	Meta-S
Undo	Meta-U

SEdit Command Menu

When the mouse cursor is in the SEdit title bar and you press middle mouse button, a Help Menu of commands pops up. The menu looks like this:

```
Commands
Abort                          M-A

Done                           C-X
Done & Compile                 C-C
Done & Close                   M-C-X
Done, Compile, & Close         M-C-C

Undo                           M-U
Redo                           M-R

Find                           M-F
Reverse Find                   M-C-F
Remove                         M-C-S
Substitute                     M-S
Find Gap                       M-N

Arglist                        M-H
Convert Comment                M-;
Edit                           M-O
Eval                           M-E
Expand                         M-X
Extract                        M-/
Inspect                        M-I
Join                           M-J
Mutate                         M-Z
Parenthesize                   M-(
Quote                          M-'

Set Print-Base                 M-B
Set Package                    M-P
Attach Menu                    M-M
```

The Help Menu lists each command and its corresponding Command Key. (C- stands for Control, M- for Meta.) The menu pops up with the mouse cursor next to the last command you used from the menu. This makes it easy to repeat a command.

SEdit Attached Menu

SEdit's Attached Command Menu contains the commonly used commands. Use the Meta-M keyboard command to bring up this menu. The menu can be closed, independently of the SEdit window. The menu looks like:

```
SEdit Command Menu
EXIT   DONE   ABORT       PAREN   QUOTE   EXTRACT
UNDO   REDO   ARGLIST     EDIT    EVAL    EXPAND
PRINT-BASE  10  PACKAGE  XCL-USER
FIND:
SUBSTITUTE:
```
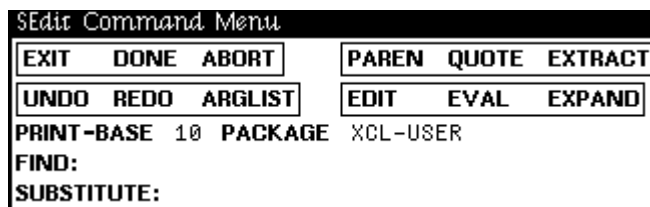
Menu commands work like the corresponding keyboard commands, except for Find and Substitute.

For Find, SEdit prompts in the menu window, next to the Find button, for the structures to find. Type in the structures then select Find again. The search begins from the caret position in the SEdit window.

Similarly, Substitute prompts next to the Find button for the structures to find, and next to the Substitute button for the structures to replace them with. After both have been typed in, selecting Substitute replaces all occurrences of the Find structures with the Substitute structures, within the current selection.

To selectively substitute, use Find to find the next potential substitution target. If you want to replace it, select Substitute. Otherwise, select Find again to go on.

Selecting either Find or Substitute with the right mouse button erases the old structure to find or substitute from the menu, and prompts for a new one.

SEdit Programmer's Interface

The following sections describe SEdit's programmer's interface.   All symbols are external in the package `SEDIT`.

SEdit Window Region Manager

SEdit provides user redefinable functions which control how SEdit chooses the region for a new edit window. In the follwin thext there are a few concepts that you will have to be familliar with. They are:

The region stack. This is a stack of old used regions. The reason to keep these around is that the user probably was comfortable with the old position of the window, so when he starts a new SEdit it is a good bet that he will be happy with the old placement.

SEdit uses   the respective value of   the symbols `SEDIT::DEFAULT-FONT`, `SEDIT::ITALIC-FONT`,   `SEDIT::KEYWORD-FONT`,   `SEDIT::COMMENT-FONT`, and `SEDIT::BROKEN-ATOM-FONT` when displaying an expression. The value of these symbols have to be font descriptors.

(**GET-WINDOW-REGION** *context reason name type*)   [Function]

This function is called when SEdit wants to know where to place a window it is about to open. This happens whenever the user starts a new SEdit or expands an Sedit icon. The default behavior is to pop a window region off SEdit's stack of regions that have been used in the past. If the stack is empty, SEdit   prompts for a new region.

context is the current editor context.

reason is one of `:CREATE` or `:EXPAND` depending on what action prompted the call to `GET-WINDOW-REGION`

name is the name of the structure to be edited.

type is the edit type of the calling context.

(**SAVE-WINDOW-REGION** *context reason name type region*)   [Function]

This function is called whenever SEdit is finished with a region and wants to make the region available for other SEdits. This happens whenever an SEdit window is closed or shrunk, or when an SEdit Icon is closed. The default behavior is simply to push the region onto SEdit's stack of regions.

context is the current editor context.

reason is one of :CLOSE, :SHRINK, or :CLOSE-ICON or depending on what action prompted the call to SAVE-WINDOW-REGION

name is the name of the structure to be edited.

type is the edit type of the calling context.

region is the region to be pushed onto the region stack. If region is NIL the old region of the SEdit will be pushed top the region stack.

KEEP-WINDOW-REGION   [Variable]

Default T. This flag determines the behavior of the default SEdit region manager, explained above, for shrinking and expanding windows. When set to T, shrinking an SEdit window will not give up that window's region; the icon will always expand back into the same region. When set to NIL, the window's region is made available for other SEdits when the window is shrunk. Then when an SEdit icon is expanded, the window will be reshaped to the next available region.

This variable is only used by the default implementations of the functions get-window-region and save-window-region. If these functions are redefined, this flag is no longer used.

Options   .

The following parameters can be set as desired.

WRAP-PARENS   [Variable]

This SEdit pretty printer flag determines whether or not trailing close parenthesis characters,), are forced to be visible in the window without scrolling. By default it is set to NIL, meaning that close parens are allowed to "fall off" the right edge of the window. If set to T, the pretty printer will start a new line before the structure preceding the close parens, so that all the parens will be visible.

WRAP-SEARCH   [Variable]

This flag determines whether or not SEdit find will wrap around to the top of the structure when it reaches the end, or vice versa in the case of reverse find. The default is NIL.

CLEAR-LINEAR-ON-COMPLETION [Variable]

This flag determines whether or not SEdit completely re-pretty prints the structure being edited when you complete the edit. The default value is NIL, meaning that SEdit reuses the pretty printing.

IGNORE-CHANGES-ON-COMPLETION [Variable]

Sometimes the structure that you are editing is changed by the system upon completion. Editdates are an example of this behavior. When this flag is NIL, the default, SEdit will redisplay the new structure, capturing the changes. When T, SEdit will ignore the fact that changes were made by the system and keep the old structure.

CONVERT-UPGRADE [Variable]

Default 100. When using Meta-; to convert old-style single- asterisk comments, if the length of the comment exceeds convert-upgrade characters, the comment is converted into a double semicolon comment. Otherwise, the comment is converted into a single semicolon comment.

Old-style double-asterisk comments are always converted into new-style triple-semicolon comments.

Control Functions

(**RESET**) [Function]

This function recomputes the SEdit edit environment. Any changes made in the font profile, or any changes made to SEdit's commands are captured by resetting. Close all SEdit windows before calling this function.

(**ADD-COMMAND** *key-code form &optional scroll? key-name command-name help-string*) [Function]

This function allows you to write your own SEdit keyboard commands. You can add commands to new keys, or you can redefine keys that SEdit already uses as command keys. If you mistakenly redefine an SEdit command, the funtion Reset-Commands will remove all user-added commands, leaving SEdit with its default set of commands.

key-code can be a character code, or any form acceptible to il:charcode.

form determines the function to be called when the key command is typed. It can be a symbol naming a function, or a list, whose first element is a symbol naming a function and the rest of the elements are extra arguments to the function. When the command is invoked, SEdit will apply the function to the edit context (SEdit's main data structure), the charcode that was typed, and any extra arguments supplied in form. The extra arguments do not get evaluated, but are useful as keywords or flags, depending on how the command was invoked. The command function must return T if it handled the command. If the function returns NIL, SEdit will ignore the command and insert the character typed.

The first optional argument, `scroll?`, determines whether or not SEdit scrolls the window after running the command. This argument defaults to `NIL`, meaning don't scroll. If the value of `scroll?` is `T`, SEdit will scroll the window to ensure that the caret is visible.

The rest of the optional arguments are used to add this command to SEdit's middle button menu. When the item is selected from the menu, the command function will be called as described above, with the charcode argument set to `NIL`.

`key-name` is a string to identify the key (combination) to be typed to invoke the command. For example "M-A" to represent the Meta-A key combination, and "C-M-A" for Control-Meta-A.

command-name is a string to identify the command function, and will appear in the menu next to the key-name.

help-string is a string to be printed in the prompt window when a mouse button is held down over the menu item.

After adding all the commands that you want, you must call Reset-Commands to install them.

For example:

```
(ADD-COMMAND "^U" (MY-CHANGE-CASE T))

(ADD-COMMAND "^Y" (MY-CHANGE-CASE NIL))

(ADD-COMMAND "1,R" MY-REMOVE-NIL
  "M-R" "REMOVE NIL"
  "REMOVE NIL FROM THE SELECTED STRUCTURE"))

(RESET-COMMANDS)
```

will add three commands.

Suppose `MY-CHANGE-CASE` takes the arguments *context*, *charcode*, and *upper-case?*. *upper-case?* will be set to `T` when `MY-CHANGE-CASE` is called from Control-U, and `NIL` when called from Control-Y. `MY-REMOVE-NIL` will be called with only *context* and *charcode* arguments when you type Meta-R.

(RESET-COMMANDS) [Function]

This function installs all commands added by add-command. SEdits which are open at the time of the reset-commands will not see the new commands; only new SEdits will have the new commands available.

(DEFAULT-COMMANDS) [Function]

This function removes all commands added by add-command, leaving SEdit with its default set of commands. As in reset-commands, open SEdits will not be changed; only new SEdits will have the user commands removed.

(**GET-PROMPT-WINDOW** *context*) [Function]

> Returns the attached prompt window for a particular SEdit.

(**GET-SELECTION** *context*) [Function]

> This function returns two values: the selected structure, and the type of selection, one of NIL, T, or :SUB-LIST. The selection type NIL means there is not a valid selection (in this case the structure is meaningless). T means the selection is one complete structure. :SUB-LIST means a series of elements in a list is selected, in which case the structure returned is a list of the elements selected.

(**REPLACE-SELECTION** *context structure selection-type*) [Function]

> This function replaces the current selection with a new structure, or multiple structures, by deleting the selection and then inserting the new structure(s). The selection-type argument must be one of T or :SUB-LIST. If T, the structure is inserted as one complete structure. If :SUB-LIST, the structure is treated as a list of elements, each of which is insertd.

EDIT-FN [Variable]

> This function is called with the selected structure and the edit specified as arguments to Sedit options as its arguments from the Edit (M-O) command. It should start the editor as appropriate, or generate an error if the selection is not editable.

COMPILE-FN [Variable]

> This function is called with the arguments *name*, *type*, and *body*, from the compile/completion commands. It should compile the definition, *body*, and install the code as appropriate.

(**SEDIT** *structure props options*) [Function]

> This function provides a means of starting SEdit directly. *structure* is the structure to be edited.

> *props* is a property list, which may specify the following properties:

> :NAME - the name of the object being edited

> :TYPE - the file manager type of the object being edited. If NIL, SEdit will not call the file manager when it tries to refetch the definition it is editing. Instead, it will just continue to use the structure that it has.

> :COMPLETION-FN - the function to be called when the edit session is completed. This function is called with the *context*, *structure*, and *changed?* arguments. *context* is SEdits main data structure. *structure* is the structure being edited. *changed?* specifies if any changes have been made, and is one of NIL, T, or :ABORT, where :ABORT means the user is aborting the edit and throwing away any changes made. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments for the function.

`:ROOT-CHANGED-FN` - the function to be called when the entire structure being edited is replaced with a new structure. This function is called with the new structure as its argument. If the value of this property is a list, the first element is treated as the function, and the rest of the elements are extra arguments that the function is applied to following the structure argument.

options is one or a list of any number of the followng keywords:

`:CLOSE-ON-COMPLETION` - This option specifies that SEdit cannot remain active for multiple completions. That is, the SEdit window cannot be shrunk, and the completion commands that normally leave the window open will in this case close the window and terminate the edit.

`:COMPILE-ON-COMPLETION` - This option specifies that SEdit should call the `*COMPILE-FN*` to compile the definition being edited upon completion, regardless of the completion command used.

The TTY Editor

This editor the main code editor in pre-window-system versions of Interlisp. For that task, it has been replaced by SEdit.

However, the TTY Editor provides an excellent language for manipulating list structure and making large-scale code changes. For example, several tools for cleaning up code are written using TTY Editor calls to do the actual work.

TTY Editor Local Attention-Changing Commands

This section describes commands that change the current expression (i.e., change the edit chain) thereby "shifting the editor's attention." These commands depend only on the *structure* of the edit chain, as compared to the search commands (presented later), which search the contents of the structure.

UP   [Editor Command]

UP modifies the edit chain so that the old current expression (i.e., the one at the time UP was called) is the first element in the new current expression. If the current expression is the first element in the next higher expression UP simply does a 0. Otherwise UP adds the corresponding tail to the edit chain.

If a P command would cause the editor to type . . . before typing the current expression, ie., the current expression is a tail of the next higher expression, UP has no effect.

For example:

```
*PP
(COND ((NULL X) (RETURN Y)))
*1 P
COND
*UP P
(COND (& &))
*-1 P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y)))
*F NULL P
(NULL X)
*UP P
((NULL X) (RETURN Y))
*UP P
... ((NULL X) (RETURN Y)))
```

The execution of UP is straightforward, except in those cases where the current expression appears more than once in the next higher expression. For example, if the current expression is (A NIL B NIL C NIL) and you perform 4 followed by UP, the current expression should then be ... NIL C NIL). UP can determine which tail is the correct one because the commands that descend save the last tail on an internal editor variable, LASTAIL. Thus after the 4 command is executed, LASTAIL is (NIL C NIL). When UP is called, it first determines if the current expression is a tail of the next higher expression. If it is, UP is finished. Otherwise, UP computes (MEMB *CURRENT-EXPRESSION* *NEXT-HIGHER-EXPRESSION*) to obtain a tail beginning with the current expression. The current expression should *always* be either a tail or an element of the next higher expression. If it is neither, for example yopu have directly (and incorrectly) manipulated the edit chain, UP generates an error. If there are no other instances of the current expression in the next higher expression, this tail is the correct one. Otherwise UP uses LASTAIL to select the correct tail.

Occasionally you can get the edit chain into a state where LASTAIL cannot resolve the ambiguity, for example if there were two non-atomic structures in the same expression that were EQ, and you descended more than one level into one of them and then tried to come back out using UP. In this case, UP prints LOCATION UNCERTAIN and generates an error. Of course, we could have solved this problem completely in our implementation by saving at each descent *both* elements and tails. However, this would be a costly solution to a situation that arises infrequently, and when it does, has no detrimental effects. The LASTAIL solution is cheap and resolves almost all of the ambiguities.

N (*N> = 1*)   [Editor Command]

Adds the *N*th element of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets LASTAIL for use by UP. Generates an error if the current expression is not a list that contains at least *N* elements.

–N (*N>* = 1)   [Editor Command]

Adds the *N*th element from the end of the current expression to the front of the edit chain, thereby making it be the new current expression. Sets `LASTAIL` for use by `UP`. Generates an error if the current expression is not a list that contains at least *N* elements.

0   [Editor Command]

Sets the edit chain to `CDR` of the edit chain, thereby making the next higher expression be the new current expression. Generates an error if there is no higher expression, i.e., `CDR` of edit chain is `NIL`.

Note that `0` usually corresponds to going back to the next higher left parenthesis, but not always. For example:

```
*P
(A B C D E F B)
*3 UP P
... C D E F G)
*3 UP P
... E F G)
*0 P
... C D E F G)
```

If the intention is to go back to the next higher left parenthesis, regardless of any intervening tails, the command `!0` can be used.

!0   [Editor Command]

Does repeated `0`'s until it reaches a point where the current expression is *not* a tail of the next higher expression, i.e., always goes back to the next higher left parenthesis.

↑   [Editor Command]

Sets the edit chain to `LAST` of edit chain, thereby making the top level expression be the current expression. Never generates an error.

NX   [Editor Command]

Effectively does an `UP` followed by a `2`, thereby making the current expression be the next expression. Generates an error if the current expression is the last one in a list. (However, `!NX` described below will handle this case.)

BK   [Editor Command]

Makes the current expression be the previous expression in the next higher expression. Generates an error if the current expression is the first expression in a list.

For example:

```
*PP
(COND ((NULL X) (RETURN Y)))
*F RETURN P
(RETURN Y)
*BK P
(NULL X)
```

Both NX and BK operate by performing a !0 followed by an appropriate number, i.e., there won't be an extra tail above the new current expression, as there would be if NX operated by performing an UP followed by a 2.

(NX *N*) [Editor Command]

(*N* >= 1) Equivalent to *N* NX commands, except if an error occurs, the edit chain is not changed.

(BK *N*) [Editor Command]

(*N* >= 1) Equivalent to *N* BK commands, except if an error occurs, the edit chain is not changed.

Note: (NX –*N*) is equivalent to (BK *N*), and vice versa.

!NX [Editor Command]

Makes the current expression be the next expression at a higher level, i.e., goes through any number of right parentheses to get to the next expression. For example:

```
*PP
(PROG ((L L)
       (UF L))
  LP  (COND
          ((NULL (SETQ L (CDR L)))
             (ERROR!))
          ([NULL (CDR (FMEMB (CAR L) (CADR L]
             (GO LP)))
        (EDITCOM (QUOTE NX))
        (SETQ UNFIND UF)
        (RETURN L))
*F CDR P
(CDR L)
*NX

NX    ?
*!NX P
(ERROR!)
*!NX P
((NULL &) (GO LP))
*!NX P
(EDITCOM (QUOTE NX))
*
```

`!NX` operates by doing `0`'s until it reaches a stage where the current expression is *not* the last expression in the next higher expression, and then does a `NX`. Thus `!NX` always goes through at least one unmatched right parenthesis, and the new current expression is always on a different level, i.e., `!NX` and `NX` always produce different results. For example using the previous current expression:

```
*F CAR P
(CAR L)
*!NX P
(GO LP)
*\P P
(CAR L)
*NX P
(CADR L)
*
```

`(NTH N)`   [Editor Command]

   `(N ~= 0)` Equivalent to *N* followed by `UP`, i.e., causes the list starting with the *N*th element of the current expression (or *N*th from the end if *N* < 0) to become the current expression. Causes an error if current expression does not have at least *N* elements.

   `(NTH 1)` is a no-op, as is `(NTH -L)` where *L* is the length of the current expression.

`line-feed`   [Editor Command]

   Moves to the "next" expression and prints it, i.e. performs a `NX` if possible, otherwise performs a `!NX`. (The latter case is indcated by first printing ">".)

`Control-X`   [Editor Command]

   Control-X moves to the "previous" thing and then prints it, i.e. performs a `BK` if possible, otherwise a `!0` followed by a `BK`.

`Control-Z`   [Editor Command]

   Control-Z moves to the last expression and prints it, i.e. does `-1` followed by `P`.

Line-feed, Control-X, and Control-Z are implemented as *immediate* read macros; as soon as they are read, they abort the current printout. They thus provide a convenient way of moving around in the editor. To facilitate using different control characters for those macros, the function `SETTERMCHARS` is provided (see below).

Commands That Search

All of the editor commands that search use the same pattern matching routine (the function `EDIT4E`, below). We will therefore begin our discussion of searching by describing the pattern match mechanism. A pattern *PAT* matches with *X* if any of the following conditions are true:

1.   If *PAT* is `EQ` to *X*

2.   If *PAT* is `&`

3.   If *PAT* is a number and `EQP` to *X*

4.   If *PAT* is a string and `(STREQUAL PAT X)` is true

5.   If `(CAR PAT)` is the atom `*ANY*`, `(CDR PAT)` is a list of patterns, and one of the patterns on `(CDR PAT)` matches *X*.

6.   If *PAT* is a literal atom or string containing one or more $s (escapes), each $ can match an indefinite number (including 0) of contiguous characters in the atom or string *X*, e.g., `VER$` matches both `VERYLONGATOM` and `"VERYLONGSTRING"` as do `$LONG$` (but not `$LONG`), and `$V$L$T$`. Note: the litatom $ (escape) matches only with itself.

7.   If *PAT* is a literal atom or string ending in $$ (escape, escape), *PAT* matches with the atom or string *X* if it is "close" to *PAT*, in the sense used by the spelling corrector (see Chapter 20). For example, `CONSS$$` matches with `CONS`, `CNONC$$` with `NCONC` or `NCONC1`.

The pattern matching routine always types a message of the form `=MATCHING-ITEM` to inform you of the object matched by a pattern of the above two types, unless `EDITQUIETFLG = T`. For example, if `VER$` matches `VERYLONGATOM`, the editor would print `=VERYLONGATOM`.

8.   If `(CAR PAT)` is the atom `--`, *PAT* matches *X* if `(CDR PAT)` matches with some tail of *X*. For example, `(A -- (&))` will match with `(A B C (D))`, but not `(A B C D)`, or `(A B C (D) E)`. However, note that `(A -- (&) --)` will match with `(A B C (D) E)`. In other words, `--` can match any interior segment of a list.

   If `(CDR PAT)`= `NIL`, i.e., *PAT* = `(--)`, then it matches any tail of a list. Therefore, `(A --)` matches `(A)`, `(A B C)` and `(A . B)`.

9.   If `(CAR PAT)` is the atom `==`, *PAT* matches *X* if and only if `(CDR PAT)` is `EQ` to *X*.

   This pattern is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command *typed* in by you obviously cannot be `EQ` to already existing structure.

10. If `(CADR PAT)` is the atom `..` (two periods), *PAT* matches *X* if `(CAR PAT)` matches `(CAR X)` and `(CDDR PAT)` is contained in *X*, as described below.

11. Otherwise if *X* is a list, *PAT* matches *X* if `(CAR PAT)` matches `(CAR X)`, and `(CDR PAT)` matches `(CDR X)`.

When the editor is searching, the pattern matching routine is called to match with *elements* in the structure, unless the pattern begins with `...` (three periods), in which case CDR of the pattern is matched against proper tails in the structure. Thus,

```
*P
        (A B C (B C))
        *F (B --)
        *P
        (B C)
        *0 F (... B --)
        *P
        ... B C (B C))
```

Matching is also attempted with atomic tails (except for NIL). Thus,

```
*P
        (A (B . C))
        *F C
        *P
        ... . C)
```

Although the current expression is the atom C after the final command, it is printed as `... . C)` to alert you to the fact that C is a *tail*, not an element. Note that the pattern C will match with either instance of C in `(A C (B . C))`, whereas `(... . C)` will match only the second C. The pattern NIL will only match with NIL as an element, i.e., it will not match in `(A B)`, even though CDDR of `(A B)` is NIL. However, `(... . NIL)` (or equivalently `(...)`) may be used to specify a NIL *tail*, e.g., `(... . NIL)` will match with CDR of the third subexpression of `((A . B) (C . D) (E))`.

Search Algorithm

Searching begins with the current expression and proceeds in print order. Searching usually means find the next instance of this pattern, and consequently a match is not attempted that would leave the edit chain unchanged. At each step, the pattern is matched against the next element in the expression currently being searched, unless the pattern begins with `...` (three periods) in which case it is matched against the next tail of the expression.

If the match is not successful, the search operation is recursive first in the CAR direction, and then in the CDR direction, i.e., if the element under examination is a list, the search descends into that list before attempting to match with other elements (or tails) at the same level. Note: A find command of the form `(F PATTERN NIL)` will only attempts matches at the top level of the current expression, i.e., it does not descend into elements, or ascend to higher expressions.

However, at no point is the total recursive depth of the search (sum of number of CARs and CDRs descended into) allowed to exceed the value of the variable MAXLEVEL. At that point, the search of that element or tail is abandoned, exactly as though the element or tail had been completely searched without finding a match, and the search continues with the element or tail for which the recursive depth is below MAXLEVEL. This feature is designed to enable you to search circular list structures (by setting MAXLEVEL small), as well as protecting him from accidentally encountering a circular list

structure in the course of normal editing. `MAXLEVEL` can also be set to `NIL`, which is equivalent to infinity. `MAXLEVEL` is initially set to 300.

If a successful match is not found in the current expression, the search automatically ascends to the next higher expression, and continues searching there on the next expression after the expression it just finished searching. If there is none, it ascends again, etc. This process continues until the entire edit chain has been searched, at which point the search fails, and an error is generated. If the search fails (or is aborted by Control-E), the edit chain is not changed (nor are any `CONS`es performed).

If the search is successful, i.e., an expression is found that the pattern matches, the edit chain is set to the value it would have had had you reached that expression via a sequence of integer commands.

If the expression that matched was a list, it will be the final link in the edit chain, i.e., the new current expression. If the expression that matched is not a list, e.g., is an atom, the current expression will be the tail beginning with that atom, unless the atom is a tail, e.g., `B` in `(A . B)`. In this case, the current expression will be `B`, but will print as `... . B)`. In other words, the search effectively does an `UP` (unless `UPFINDFLG = NIL` (initially `T`). See "Form Oriented Editing" in this chapter).

Search Commands

All of the commands below set `LASTAIL` for use by `UP`, set `UNFIND` for use by `\` (below), and do not change the edit chain or perform any `CONS`es if they are unsuccessful or aborted.

F *PATTERN*   [Editor Command]

> Actually two commands: the `F` informs the editor that the *next* command is to be interpreted as a pattern. This is the most common and useful form of the find command. If successful, the edit chain always changes, i.e., `F` *PATTERN* means find the next instance of *PATTERN*.

> If `(MEMB PATTERN  CURRENT-EXPRESSION)` is true, `F` does not proceed with a full recursive search. If the value of the `MEMB` is `NIL`, `F` invokes the search algorithm described above.

> If the current expression is `(PROG NIL LP (COND (-- (GO LP1))) ... LP1 ...)`, then `F  LP1` will find the `PROG` label, not the `LP1` inside of the `GO` expression, even though the latter appears first (in print order) in the current expression. Typing `1` (making the atom `PROG` be the current expression) followed by `F  LP1` *would* find the first `LP1`.

F *PATTERN* N   [Editor Command]

> Same as `F` *PATTERN*, i.e., Finds the Next instance of *PATTERN*, except that the `MEMB` check of `F` *PATTERN* is not performed.

`F` *PATTERN* `T`   [Editor Command]

Similar to `F` *PATTERN*, except that it may succeed without changing the edit chain, and it does not perform the `MEMB` check. For example, if the current expression is (`COND ...`), `F COND` will look for the next `COND`, but (`F COND T`) will "stay here".

(`F` *PATTERN* `N`)   [Editor Command]

(*N* `>= 1`) Finds the *N*th place that *PATTERN* matches. Equivalent to (`F` *PATTERN* `T`) followed by (`F` *PATTERN* `N`) repeated *N*-1 times. Each time *PATTERN* successfully matches, *N* is decremented by 1, and the search continues, until *N* reaches 0. Note that *PATTERN* does not have to match with *N* identical expressions; it just has to match *N* times. Thus if the current expression is (`FOO1 FOO2 FOO3`), (`F FOO$ 3`) will find `FOO3`.

If *PATTERN* does not match successfully *N* times, an error is generated and the edit chain is unchanged (even if *PATTERN* matched *N*-1 times).

(`F` *PATTERN*)   [Editor Command]
`F` *PATTERN* `NIL`   [Editor Command]

Similar to `F` *PATTERN*, except that it only matches with elements at the top level of the current expression, i.e., the search will not descend into the current expression, nor will it go outside of the current expression. May succeed without changing the edit chain.

For example, if the current expression is (`PROG NIL (SETQ X (COND & &)) (COND &) ...`), the command `F COND` will find the `COND` inside the `SETQ`, whereas (`F (COND --)`) will find the top level `COND`, i.e., the second one.

(`FS` *PATTERN*$_1$ `...` *PATTERN*$_N$)   [Editor Command]

Equivalent to `F` *PATTERN*$_1$ followed by `F` *PATTERN*$_2$ `...` followed by `F` *PATTERN*$_N$, so that if `F` *PATTERN*$_M$ fails, the edit chain is left at the place *PATTERN*$_{M-1}$ matched.

(`F=` *EXPRESSION* *X*)   [Editor Command]

Equivalent to (`F (== . `*EXPRESSION*`) `*X*), i.e., searches for a structure `EQ` to *EXPRESSION* (see above).

(`ORF` *PATTERN*$_1$ `...` *PATTERN*$_N$)   [Editor Command]

Equivalent to (`F (*ANY* `*PATTERN*$_1$ `...` *PATTERN*$_N$) `N`), i.e., searches for an expression that is matched by either *PATTERN*$_1$, *PATTERN*$_2$, `...` or *PATTERN*$_N$ (see above).

BF *PATTERN* [Editor Command]

"Backwards Find". Searches in reverse print order, beginning with the expression immediately before the current expression (unless the current expression is the top level expression, in which case BF searches the entire expression, in reverse order).

BF uses the same pattern match routine as F, and MAXLEVEL and UPFINDFLG have the same effect, but the searching begins at the *end* of each list, and descends into each element before attempting to match that element. If unsuccessful, the search continues with the next previous element, etc., until the front of the list is reached, at which point BF ascends and backs up, etc.

For example, if the current expression is

(PROG NIL (SETQ X (SETQ Y (LIST Z))) (COND ((SETQ W --) --)) --),

the command F LIST followed by BF SETQ will leave the current expression as (SETQ Y (LIST Z)), as will F COND followed by BF SETQ.

BF *PATTERN* T [Editor Command]

Similar to BF *PATTERN*, except that the search always includes the current expression, i.e., starts at the end of current expression and works backward, then ascends and backs up, etc.

Thus in the previous example, where F COND followed by BF SETQ found (SETQ Y (LIST Z)), F COND followed by (BF SETQ T) would find the (SETQ W --) expression.

(BF *PATTERN*) [Editor Command]
BF *PATTERN* NIL [Editor Command]

Same as BF *PATTERN*.

(GO *LABEL*) [Editor Command]

Makes the current expression be the first thing after the PROG label *LABEL*, i.e. goes where an executed GO would go.

Location Specification

Many of the more sophisticated commands described later in this chapter use a more general method of specifying position called a "location specification." A location specification is a list of edit commands that are executed in the normal fashion with two exceptions. First, all commands not recognized by the editor are interpreted as though they had been preceded by F; normally such commands would cause errors. For example, the location specification (COND 2 3) specifies the 3rd element in the first clause of the next COND. Note that you could always write F COND followed by 2

and 3 for `(COND 2 3)` if you were not sure whether or not `COND` was the name of an atomic command.

Secondly, if an error occurs while evaluating one of the commands in the location specification, and the edit chain had been changed, i.e., was not the same as it was at the beginning of that execution of the location specification, the location operation will continue. In other words, the location operation keeps going unless it reaches a state where it detects that it is "looping", at which point it gives up. Thus, if `(COND 2 3)` is being located, and the first clause of the next `COND` contained only two elements, the execution of the command 3 would cause an error. The search would then continue by looking for the next `COND`. However, if a point were reached where there were no further `COND`s, then the first command, `COND`, would cause the error; the edit chain would not have been changed, and so the entire location operation would fail, and cause an error.

The `IF` command (see above) in conjunction with the `##` function (see below) provide a way of using arbitrary predicates applied to elements in the current expression. `IF` and `##` will be described in detail later in the chapter, along with examples illustrating their use in location specifications.

Throughout this chapter, the meta-symbol `@` is used to denote a location specification. Thus `@` is a list of commands interpreted as described above. `@` can also be atomic, in which case it is interpreted as `(LIST @)`.

`(LC . @)`   [Editor Command]

  Provides a way of explicitly invoking the location operation, e.g., `(LC COND 2 3)` will perform the the search described above.

`(LCL . @)`   [Editor Command]

  Same as `LC` except the search is confined to the current expression, i.e., the edit chain is rebound during the search so that it looks as though the editor were called on just the current expression. For example, to find a `COND` containing a `RETURN`, one might use the location specification `(COND (LCL RETURN) \)` where the `\` would reverse the effects of the `LCL` command, and make the final current expression be the `COND`.

`(2ND . @)`   [Editor Command]

  Same as `(LC . @)` followed by another `(LC . @)` except that if the first succeeds and second fails, no change is made to the edit chain.

`(3ND . @)`   [Editor Command]

  Similar to `2ND`.

`(← PATTERN)`   [Editor Command]

  Ascends the edit chain looking for a link which matches *PATTERN*. In other words, it keeps doing `0`'s until it gets to a specified point. If *PATTERN* is atomic, it is matched with the first

element of each link, otherwise with the entire link. If no match is found, an error is generated, and the edit chain is unchanged.

If *PATTERN* is of the form (IF *EXPRESSION*), *EXPRESSION* is evaluated at each link, and if its value is NIL, or the evaluation causes an error, the ascent continues.

For example:

```
*PP
[PROG NIL
    (COND
        [(NULL (SETQ L (CDR L)))
          (COND
             (FLG (RETURN L]
        ([NULL (CDR (FMEMB (CAR L)
          (CADR L]]
*F CADR
*(← COND)
*P
(COND (& &) (& &))
*
```

Note that this command differs from BF in that it does not search *inside* of each link, it simply ascends. Thus in the above example, F CADR followed by BF COND would find (COND (FLG (RETURN L))), not the higher COND.

(BELOW *COM X*)   [Editor Command]

Ascends the edit chain looking for a link specified by *COM*, and stops *X* links below that (only links that are elements are counted, not tails). In other words BELOW keeps doing 0's until it gets to a specified point, and then backs off *X* 0's.

Note that *X* is evaluated, so one can type (BELOW *COM* (IPLUS X Y)).

(BELOW *COM*)   [Editor Command]

Same as (BELOW *COM* 1).

For example, (BELOW COND) will cause the COND *clause* containing the current expression to become the new current expression. Thus if the current expression is as shown above, F CADR followed by (BELOW COND) will make the new expression be ([NULL (CDR (FMEMB (CAR L) (CADR L] (GO LP)), and is therefore equivalent to 0 0 0 0.

The BELOW command is useful for locating a substructure by specifying something it contains. For example, suppose you are editing a list of lists, and want to find a sublist that contains a FOO (at any depth). He simply executes F FOO (BELOW \).

(NEX *COM*)   [Editor Command]

Same as (BELOW *COM*) followed by NX.

For example, if you are deep inside of a SELECTQ clause, you can advance to the next clause
with (NEX SELECTQ).

NEX   [Editor Command]

Same as (NEX ←).

The atomic form of NEX is useful if you will be performing repeated executions of (NEX
COM). By simply MARKing (see the next section) the chain corresponding to *COM*, you can use
NEX to step through the sublists.

(NTH *COM*)   [Editor Command]

Generalized NTH command. Effectively performs (LCL . *COM*), followed by (BELOW \),
followed by UP.

If the search is unsuccessful, NTH generates an error and the edit chain is not changed.

Note that (NTH *NUMBER*) is just a special case of (NTH *COM*), and in fact, no special check is
made for *COM* a number; both commands are executed identically.

In other words, NTH locates *COM*, using a search restricted to the current expression, and then
backs up to the current level, where the new current expression is the tail whose first element
contains, however deeply, the expression that was the terminus of the location operation.
For example:

```
*P
(PROG (& &) LP (COND & &) (EDITCOM &) (SETQ UNFIND
UF) (RETURN L))
*(NTH UF)
*P
... (SETQ UNFIND UF) (RETURN L))
*
```

PATTERN .. @   [Editor Command]

For example, (COND .. RETURN). Finds a COND that contains a RETURN, at any depth.
Equivalent to (but more efficient than) (F *PATTERN* N), (LCL . @) followed by (←
PATTERN).

An infix command, ".." is not a meta-symbol, it *is* the name of the command. @ is CDDR of the command. Note that `(PATTERN .. @)` can also be used directly as an edit pattern as described above, e.g. F `(PATTERN .. @)`.

For example, if the current expression is

```
(PROG NIL [COND ((NULL L) (COND (FLG (RETURN L] --),
```

then `(COND .. RETURN)` will make `(COND (FLG (RETURN L)))` be the current expression. Note that it is the innermost COND that is found, because this is the first COND encountered when ascending from the RETURN. In other words, `(PATTERN .. @)` is not *always* equivalent to `(F PATTERN N)`, followed by `(LCL . @)` followed by \.

Note that @ is a location specification, not just a pattern. Thus `(RETURN .. COND 2 3)` can be used to find the RETURN which contains a COND whose first clause contains (at least) three elements. Note also that since @ permits any edit command, you can write commands of the form `(COND .. (RETURN .. COND))`, which will locate the first COND that contains a RETURN that contains a COND.

Commands That Save and Restore the Edit Chain

Several facilities are available for saving the current edit chain and later retrieving it: MARK, which marks the current chain for future reference, ←, which returns to the last mark without destroying it, and ←←, which returns to the last mark and also erases it.

MARK   [Editor Command]

Adds the current edit chain to the front of the list MARKLST.

←   [Editor Command]

Makes the new edit chain be `(CAR MARKLST)`. Generates an error if MARKLST is NIL, i.e., no MARKs have been performed, or all have been erased.

This is an atomic command; do not confuse it with the list command `(← PATTERN)`.

←←   [Editor Command]

Similar to ← but also erases the last MARK, i.e., performs `(SETQ MARKLST (CDR MARKLST))`.

If you have two chains marked, and wish to return to the first chain, you must perform ←←, which removes the second mark, and then ←. However, the second mark is then no longer

accessible. If you want to be able to return to either of two (or more) chains, you can use the following generalized MARK:

(MARK *SYMBOL*)   [Editor Command]

    Sets *SYMBOL* to the current edit chain,

(\ *SYMBOL*)   [Editor Command]

    Makes the current edit chain become the value of *SYMBOL*.

    If you did not prepare in advance for returning to a particular edit chain, you may still be able to return to that chain with a single command by using \ or \P.

\   [Editor Command]

    Makes the edit chain be the value of UNFIND. Generates an error if UNFIND = NIL.

    UNFIND is set to the current edit chain by each command that makes a "big jump", i.e., a command that usually performs more than a single ascent or descent, namely ↑, ←, ←←, !NX, all commands that involve a search, e.g., F, LC, **..**, BELOW, et al and \ and \P themselves. One exception is that UNFIND is not reset when the current edit chain is the top level expression, since this could always be returned to via the ↑ command.

    For example, if you type F COND, and then F CAR, \ would take you back to the COND. Another \ would take you back to the CAR, etc.

\P   [Editor Command]

    Restores the edit chain to its state as of the last print operation, i.e., P, ?, or PP. If the edit chain has not changed since the last printing, \P restores it to its state as of the printing before that one, i.e., two chains are always saved.

    For example, if you type P followed by 3 2 1 P, \P returns to the first P, i.e., would be equivalent to 0 0 0. Another \P would then take you back to the second P, i.e., you could use \P to flip back and forth between the two edit chains.

    If you had typed P followed by F COND, you could use *either* \ or \P to return to the P, i.e., the action of \ and \P are independent.

S *SYMBOL* @   [Editor Command]

    Sets *SYMBOL* (using SETQ) to the current expression after performing (LC . @). The edit chain is not changed.

Thus `(S FOO)` will set `FOO` to the current expression, and `(S FOO -1 1)` will set `FOO` to the first element in the last element of the current expression.

Commands That Modify Structure

The basic structure modification commands in the editor are:

`(N) (N >= 1)` [Editor Command]

    Deletes the corresponding element from the current expression.

`(N E`$_1$` ... E`$_M$`) (N >= 1)` [Editor Command]

    Replaces the Nth element in the current expression with E_1 ... E_M.

`(-N E`$_1$` ... E`$_M$`) (N >= 1)` [Editor Command]

    Inserts E_1 ... E_M before the Nth element in the current expression.

`(N E`$_1$` ... E`$_M$`)` [Editor Command]

    Attaches E_1 ... E_M at the end of the current expression.

As mentioned earlier: *all structure modification done by the editor is destructive, i.e., the editor uses* `RPLACA` *and* `RPLACD` *to physically change the structure it was given.* However, all structure modification is undoable, see `UNDO`.

All of the above commands generate errors if the current expression is not a list, or in the case of the first three commands, if the list contains fewer than N elements. In addition, the command `(1)`, i.e., delete the first element, will cause an error if there is only one element, since deleting the first element must be done by replacing it with the second element, and then deleting the second element. Or, to look at it another way, deleting the first element when there is only one element would require changing a list to an atom (i.e., to `NIL`) which cannot be done. However, the command `DELETE` will work even if there is only one element in the current expression, since it will ascend to a point where it *can* do the deletion.

If the value of `CHANGESARRAY` is a hash array, the editor will mark all structures that are changed by doing `(PUTHASH STRUCTURE FN CHANGESARRAY)`, where `FN` is the name of the function. The algorithm used for marking is as follows:

    1.   If the expression is inside of another expression already marked as being changed, do nothing.

2. If the change is an insertion of or replacement with a list, mark the list as changed.

3. If the change is an insertion of or replacement with an atom, or a deletion, mark the parent as changed.

CHANGESARRAY is primarily for use by PRETTYPRINT (Chapter 26). When the value of CHANGECHAR is non-NIL, PRETTYPRINT, when printing to a file or display terminal, prints CHANGECHAR in the right margin while printing an expression marked as having been changed. CHANGECHAR is initially |.

Implementation

> *Note: Since all commands that insert, replace, delete or attach structure use the same low level editor functions, the remarks made here are valid for all structure changing commands.*

For all replacement, insertion, and attaching at the end of a list, unless the command was typed in directly to the editor, *copies* of the corresponding structure are used, because of the possibility that the exact same command, (i.e., same list structure) might be used again. Thus if a program constructs the command (1 (A B C)) e.g., via (LIST 1 FOO), and gives this command to the editor, the (A B C) used for the replacement will *not* be EQ to FOO. You can circumvent this by using the I command, which computes the structure to be used. In the above example, the form of the command would be (I 1 FOO), which would replace the first element with the value of FOO itself.

> Note: Some editor commands take as arguments a list of edit commands, e.g., (LP F FOO (1 (CAR FOO))). In this case, the command (1 (CAR FOO)) is not considered to have been "typed in" even though the LP command itself may have been typed in. Similarly, commands originating from macros, or commands given to the editor as arguments to EDITF, EDITV, et al, e.g., EDITF(FOO F COND (N --)) are not considered typed in.

The rest of this section is included for applications wherein the editor is used to modify a data structure, and pointers into that data structure are stored elsewhere. In these cases, the actual mechanics of structure modification must be known in order to predict the effect that various commands may have on these outside pointers. For example, if the value of FOO is CDR of the current expression, what will the commands (2), (3), (2 X Y Z), (-2 X Y Z), etc. do to FOO?

Deletion of the first element in the current expression is performed by replacing it with the second element and deleting the second element by patching around it. Deletion of any other element is done by patching around it, i.e., the previous tail is altered. Thus if FOO is EQ to the current expression which is (A B C D), and FIE is CDR of FOO, after executing the command (1), FOO will be (B C D) (which is EQUAL but not EQ to FIE). However, under the same initial conditions, after executing (2) FIE will be unchanged, i.e., FIE will still be (B C D) even though the current expression and FOO are now (A C D).

A general solution of the problem isn't possible, as it would require being able to make two lists EQ to each other that were originally different. Thus if FIE is CDR of the current expression, and FUM is CDDR of the current expression, performing (2) would have to make FIE be EQ to FUM if all subsequent operations were to update both FIE and FUM correctly.

Both replacement and insertion are accomplished by smashing both `CAR` and `CDR` of the corresponding tail. Thus, if `FOO` were `EQ` to the current expression, `(A B C D)`, after `(1 X Y Z)`, `FOO` would be `(X Y Z B C D)`. Similarly, if `FOO` were `EQ` to the current expression, `(A B C D)`, then after `(-1 X Y Z)`, `FOO` would be `(X Y Z A B C D)`.

The `N` command is accomplished by smashing the last `CDR` of the current expression a la `NCONC`. Thus if `FOO` were `EQ` to any tail of the current expression, after executing an `N` command, the corresponding expressions would also appear at the end of `FOO`.

In summary, the only situation in which an edit operation will *not* change an external pointer occurs when the external pointer is to a *proper tail* of the data structure, i.e., to `CDR` of some node in the structure, and the operation is deletion. If all external pointers are to *elements* of the structure, i.e., to `CAR` of some node, or if only insertions, replacements, or attachments are performed, the edit operation will *always* have the same effect on an external pointer as it does on the current expression.

The A, B, and : Commands

In the `(N)`, `(N E_1 ... E_M)`, and `(-N E_1 ... E_M)` commands, the sign of the integer is used to indicate the operation. As a result, there is no direct way to express insertion after a particular element (hence the necessity for a separate `N` command). Similarly, you cannot specify deletion or replacement of the Nth element from the end of a list without first converting N to the corresponding positive integer. Accordingly, we have:

`(B E_1 ... E_M)`   [Editor Command]

Inserts $E_1 ... E_M$ before the current expression. Equivalent to `UP` followed by `(-1 E_1 ... E_M)`.

For example, to insert `FOO` before the last element in the current expression, perform `-1` and then `(B FOO)`.

`(A E_1 ... E_M)`   [Editor Command]

Inserts $E_1 ... E_M$ after the current expression. Equivalent to `UP` followed by `(-2 E_1 ... E_M)` or `(N E_1 ... E_M)`, whichever is appropriate.

`(: E_1 ... E_M)`   [Editor Command]

Replaces the current expression by $E_1 ... E_M$. Equivalent to `UP` followed by `(1 E_1 ... E_M)`.

```
DELETE                                                          [Editor Command]
(:)                                                             [Editor Command]
```

Deletes the current expression.

DELETE first tries to delete the current expression by performing an UP and then a (1). This works in most cases. However, if after performing UP, the new current expression contains only one element, the command (1) will not work. Therefore, DELETE starts over and performs a BK, followed by UP, followed by (2). For example, if the current expression is (COND ((MEMB X Y)) (T Y)), and you perform -1, and then DELETE, the BK-UP-(2) method is used, and the new current expression will be ... ((MEMB X Y))).

However, if the next higher expression contains only one element, BK will not work. So in this case, DELETE performs UP, followed by (: NIL), i.e., it *replaces* the higher expression by NIL. For example, if the current expression is (COND ((MEMB X Y)) (T Y)) and you perform F MEMB and then DELETE, the new current expression will be ... NIL (T Y)) and the original expression would now be (COND NIL (T Y)). The rationale behind this is that deleting (MEMB X Y) from ((MEMB X Y)) changes a list of one element to a list of no elements, i.e., () or NIL.

If the current expression is a tail, then B, A, :, and DELETE all work exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (B (PRINT X)) would insert (PRINT X) before (PRINT Y), leaving the current expression ... (PRINT X) (PRINT Y) (PRINT Z)).

The following forms of the A, B, and : commands incorporate a location specification:

```
(INSERT E₁ ... E_M BEFORE . @)                                 [Editor Command]
```

(@ is (CDR (MEMBER 'BEFORE *COMMAND*))) Similar to (LC .@) followed by (B E₁ ... E_M).

> Warning: If @ causes an error, the location process does *not* continue as described above. For example, if @ = (COND 3) and the next COND does not have a thirdelement, the search stops and the INSERT fails. You can always write (LC COND 3) if you intend the search to continue.
>
> ```
> *P
> (PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR &
> &) (PRIN1 & T)
> (PRIN1 & T) (SETQ X &
>
> *(INSERT LABEL BEFORE PRIN1)
> *P
> ```

```
(PROG (& & X) **COMMENT** (SELECTQ ATM & NIL) (OR &
&) LABEL
(PRIN1 & T) (          user typed Control-E

*
```

Current edit chain is not changed, but `UNFIND` is set to the edit chain after the `B` was performed, i.e., \ will make the edit chain be that chain where the insertion was performed.

(INSERT E_1 ... E_M AFTER . @)   [Editor Command]

Similar to INSERT BEFORE except uses A instead of B.

(INSERT E_1 ... E_M FOR . @)   [Editor Command]

Similar to INSERT BEFORE except uses : for B.

(REPLACE @ BY E_1 ... E_M)   [Editor Command]
(REPLACE @ WITH E_1 ... E_M)   [Editor Command]

Here @ is the *segment* of the command between REPLACE and WITH. Same as (INSERT E_1 ... E_M FOR . @).

Example: (REPLACE COND -1 WITH (T (RETURN L)))

(CHANGE @ TO E_1 ... E_M)   [Editor Command]

Same as REPLACE WITH.

(DELETE . @)   [Editor Command]

Does a (LC . @) followed by DELETE (see warning about INSERT above). The current edit chain is not changed, but UNFIND is set to the edit chain after the DELETE was performed.

> Note: The edit chain will be changed if the current expression is no longer a part of the expression being edited, e.g., if the current expression is ... C) and you perform (DELETE 1), the tail, (C), will have been cut off. Similarly, if the current expression is (CDR Y) and you perform (REPLACE WITH (CAR X)).

Example: (DELETE -1), (DELETE COND 3)

> *Note: If @ is* NIL *(i.e., empty), the corresponding operation is performed on the current edit chain.*

For example, `(REPLACE WITH (CAR X))` is equivalent to `(: (CAR X))`. For added readability, `HERE` is also permitted, e.g., `(INSERT (PRINT X) BEFORE HERE)` will insert `(PRINT X)` before the current expression (but not change the edit chain).

Note: @ *does not have to specify a location within the current expression, i.e., it is perfectly legal to ascend to* `INSERT`, `REPLACE`, *or* `DELETE`

For example, `(INSERT (RETURN) AFTER ^ PROG -1)` will go to the top, find the first `PROG`, and insert a `(RETURN)` at its end, and not change the current edit chain.

The `A`, `B`, and `:` commands, commands, (and consequently `INSERT`, `REPLACE`, and `CHANGE`), all make special checks in E_1 thru E_M for expressions of the form `(## . COMS)`. In this case, the expression used for inserting or replacing is a *copy* of the current expression after executing *COMS*, a list of edit commands (the execution of *COMS* does not change the current edit chain). For example, `(INSERT (## F COND -1 -1) AFTER 3)` will make a copy of the last form in the last clause of the next `COND`, and insert it after the third element of the current expression. Note that this is not the same as `(INSERT F COND -1 (## -1) AFTER 3)`, which inserts four elements after the third element, namely `F`, `COND`, `-1`, and a copy of the last element in the current expression.

Form Oriented Editing and the Role of UP

The `UP` that is performed before `A`, `B`, and `:` commands (and therefore in `INSERT`, `CHANGE`, `REPLACE`, and `DELETE` commands after the location portion of the operation has been performed) makes these operations form-oriented. For example, if you type `F SETQ`, and then `DELETE`, or simply `(DELETE SETQ)`, you will delete the entire `SETQ` expression, whereas `(DELETE X)` if `X` is a variable, deletes just the variable `X`. In both cases, the operation is performed on the corresponding *form*, and in both cases is probably what you intended. Similarly, if you type `(INSERT (RETURN Y) BEFORE SETQ)`, you mean before the `SETQ` expression, not before the atom `SETQ`. A consequent of this procedure is that a pattern of the form `(SETQ Y --)` can be viewed as simply an elaboration and further refinement of the pattern `SETQ`. Thus `(INSERT (RETURN Y) BEFORE SETQ)` and `(INSERT (RETURN Y) BEFORE (SETQ Y --))` perform the same operation (assuming the next `SETQ` is of the form `(SETQ Y --))` and, in fact, this is one of the motivations behind making the current expression after `F SETQ`, and `F (SETQ Y --)` be the same.

Note: There is some ambiguity in `(INSERT EXPR AFTER FUNCTIONNAME)`, as you might mean make *EXPR* be the function's first argument. Similarly, you cannot write `(REPLACE SETQ WITH SETQQ)` meaning change the name of the function. You must in these cases write `(INSERT EXPR AFTER FUNCTIONNAME 1)`, and `(REPLACE SETQ 1 WITH SETQQ)`.

Occasionally, however, you may have a data structure in which no special significance or meaning is attached to the position of an atom in a list, as Interlisp attaches to atoms that appear as `CAR` of a list, versus those appearing elsewhere in a list. In general, you may not even *know* whether a particular atom is at the head of a list or not. Thus, when you write `(INSERT EXPR BEFORE FOO)`, you mean before the atom `FOO`, whether or not it is `CAR` of a list. By setting the variable `UPFINDFLG` to `NIL`

(initially `T`), you can suppress the implicit `UP` that follows searches for atoms, and thus achieve the desired effect. With `UPFINDFLG = NIL`, following `F  FOO`, for example, the current expression will be the atom `FOO`. In this case, the `A`, `B`, and `:` operations will operate with respect to the atom `FOO`. If you intend the operation to refer to the list which `FOO` heads, use the pattern `(FOO  --)` instead .

Extract and Embed

Extraction involves replacing the current expression with one of its subexpressions (from any depth).

`(XTR . @)`   [Editor Command]

> Replaces the original current expression with the expression that is current after performing `(LCL . @)` (see warning about `INSERT` above). If the current expression after `(LCL . @)` is a *tail* of a higher expression, its first element is used.
>
> If the extracted expression is a list, then after `XTR` has finished, the current expression will be that list. If the extracted expression is not a list, the new current expression will be a tail whose first element is that non-list.
>
> For example, if the current expression is `(COND ((NULL X) (PRINT Y)))`, `(XTR PRINT)`, or `(XTR 2 2)` will replace the `COND` by the `PRINT`. The current expression after the `XTR` would be `(PRINT Y)`.
>
> If the current expression is `(COND ((NULL X) Y) (T Z))`, then `(XTR Y)` will replace the `COND` with `Y`, even though the current expression after performing `(LCL Y)` is `... Y)`. The current expression after the `XTR` would be `... Y` followed by whatever followed the `COND`.
>
> If the current expression *initially* is a tail, extraction works exactly the same as though the current expression were the first element in that tail. Thus if the current expression is `... (COND ((NULL X) (PRINT Y))) (RETURN Z))`, then `(XTR PRINT)` will replace the `COND` by the `PRINT`, leaving `(PRINT Y)` as the current expression.

The extract command can also incorporate a location specification:

`(EXTRACT @1 FROM . @2)`   [Editor Command]

> Performs `(LC . @2)` and then `(XTR . @1)` (see warning about `INSERT`). The current edit chain is not changed, but `UNFIND` is set to the edit chain after the `XTR` was performed.
>
> > Note: $@_1$ is the *segment* between `EXTRACT` and `FROM`.
>
> For example: If the current expression is `(PRINT (COND ((NULL X) Y) (T Z)))` then following `(EXTRACT  Y  FROM  COND)`, the current expression will be `(PRINT  Y)`. `(EXTRACT 2 -1 FROM COND)`, `(EXTRACT Y FROM 2)`, and `(EXTRACT 2 -1 FROM 2)` will all produce the same result.

While extracting replaces the current expression by a subexpression, embedding replaces the current expression with one containing *it* as a subexpression.

(MBD E_1 ... E_M)   [Editor Command]

MBD substitutes the current expression for all instances of the atom & in E_1 ... E_M, and replaces the current expression with the result of that substitution. As with SUBST, a fresh copy is used for each substitution.

If & does not appear in E_1 ... E_M, the MBD is interpreted as (MBD (E_1 ... E_M &)).

MBD leaves the edit chain so that the larger expression is the new current expression.

Examples:

If the current expression is (PRINT Y), (MBD (COND ((NULL X) &) ((NULL (CAR Y)) & (GO LP)))) would replace (PRINT Y) with (COND ((NULL X) (PRINT Y)) ((NULL (CAR Y)) (PRINT Y) (GO LP))).

If the current expression is (RETURN X), (MBD (PRINT Y) (AND FLG &)) would replace it with the *two* expressions (PRINT Y) and (AND FLG (RETURN X)), i.e., if the (RETURN X) appeared in the cond clause (T (RETURN X)), after the MBD, the clause would be (T (PRINT Y) (AND FLG (RETURN X))).

If the current expression is (PRINT Y), then (MBD SETQ X) will replace it with (SETQ X (PRINT Y)). If the current expression is (PRINT Y), (MBD RETURN) will replace it with (RETURN (PRINT Y)).

If the current expression *initially* is a tail, embedding works exactly the same as though the current expression were the first element in that tail. Thus if the current expression were ... (PRINT Y) (PRINT Z)), (MBD SETQ X) would replace (PRINT Y) with (SETQ X (PRINT Y)).

The embed command can also incorporate a location specification:

(EMBED @ IN . X)   [Editor Command]

(@ is the segment between EMBED and IN.) Does (LC . @) and then (MBD . X) (see warning about INSERT). Edit chain is not changed, but UNFIND is set to the edit chain after the MBD was performed.

Examples: (EMBED PRINT IN SETQ X), (EMBED 3 2 IN RETURN), (EMBED COND 3 1 IN (OR & (NULL X))).

WITH can be used for IN, and SURROUND can be used for EMBED, e.g., (SURROUND NUMBERP WITH (AND & (MINUSP X))).

EDITEMBEDTOKEN [Variable]

The special atom used in the MBD and EMBED commands is the value of this variable, initially &.

The MOVE Command

The MOVE command allows you to specify the expression to be moved, the place it is to be moved to, and the operation to be performed there, e.g., insert it before, insert it after, replace, etc.

(MOVE @$_1$ TO *COM* . @$_2$) [Editor Command]

(@$_1$ is the segment between MOVE and TO.) *COM* is BEFORE, AFTER, or the name of a list command, e.g., :, N, etc. Performs (LC . @$_1$) (see warning about INSERT), and obtains the current expression there (or its first element, if it is a tail), which we will call *EXPR*; MOVE then goes back to the original edit chain, performs (LC . @$_2$) followed by (*COM EXPR*) (setting an internal flag so *EXPR* is not copied), then goes back to @$_1$ and deletes *EXPR*. The edit chain is not changed. UNFIND is set to the edit chain after (*COM EXPR*) was performed.

If @$_2$ specifies a location *inside of the expression to be moved,* a message is printed and an error is generated, e.g., (MOVE 2 TO AFTER X), where X is contained inside of the second element.

For example, if the current expression is (A B C D), (MOVE 2 TO AFTER 4) will make the new current expression be (A C D B). Note that 4 was executed as of the original edit chain, and that the second element had not yet been removed.

As the following examples taken from actual editing will show, the MOVE command is an extremely versatile and powerful feature of the editor.

```
*?
(PROG ((L L)) (EDLOC (CDDR C)) (RETURN (CAR L)))
*(MOVE 3 TO : CAR)
*?
(PROG ((L L)) (RETURN (EDLOC (CDDR C))))
*
*P
... (SELECTQ OBJPR & &) (RETURN &) LP2 (COND & &))
*(MOVE 2 TO N 1)
*P
... (SELECTQ OBJPR & & &) LP2 (COND & &))


*
*P
```

```
                    (OR (EQ X LASTAIL) (NOT &) (AND & & &))
                    *(MOVE 4 TO AFTER (BELOW COND))
                    *P
                    (OR (EQ X LASTAIL) (NOT &))
                    *\ P
                    ... (& &) (AND & & &) (T & &))
                    *

                    *P
                    ((NULL X) **COMMENT** (COND & &))
                    *(-3 (GO NXT]
                    *(MOVE 4 TO N (← PROG))
                    *P
                    ((NULL X) **COMMENT** (GO NXT))
                    *\ P
                    (PROG (&) **COMMENT** (COND & & &) (COND & & &) (COND & &))
                    *(INSERT NXT BEFORE -1)
                    *P
                    (PROG (&) **COMMENT** (COND & & &) (COND & & &) NXT (COND & &))
```

In the last example, you could have added the PROG label NXT and moved the COND in one operation by performing (MOVE 4 TO N (← PROG) (N NXT)). Similarly, in the next example, in the course of specifying $@_2$, the location where the expression was to be moved to, you also perform a structure modification, via (N (T)), thus creating the structure that will receive the expression being moved.

```
                    *P
                    ((CDR &) **COMMENT** (SETQ CL &) (EDITSMASH CL & &))
                    *MOVE 4 TO N 0 (N (T)) -1]
                    *P
                    ((CDR &) **COMMENT** (SETQ CL &))
                    *\ P
                    *(T (EDITSMASH CL & &))
                    *
```

If $@_2$ is NIL, or (HERE), the current position specifies where the operation is to take place. In this case, UNFIND is set to where the expression that was moved was originally located, i.e., $@_1$. For example:

```
                    *P
                    (TENEX)
                    *(MOVE ↑ F APPLY TO N HERE)
                    *P
                    (TENEX (APPLY & &))
                    *

                    *P
                    (PROG (& & & ATM IND VAL) (OR & &) **COMMENT** (OR & &)
                    (PRIN1 & T) (
                    PRIN1 & T) (SETQ IND    user typed Control-E

                    *(MOVE * TO BEFORE HERE)
                    *P
                    (PROG (& & & ATM IND VAL) (OR & &) (OR & &) (PRIN1 &
```

```
*P
(T (PRIN1 C-EXP T))
*(MOVE ↑ BF PRIN1 TO N HERE)
*P
(T (PRIN1 C-EXP T) (PRIN1 & T))
*
```

Finally, if @$_1$ is NIL, the MOVE command allows you to specify where the *current expression* is to be moved to. In this case, the edit chain is changed, and is the chain where the current expression was moved to; UNFIND is set to where it was.

```
*P
(SELECTQ OBJPR (&) (PROGN & &))
*(MOVE TO BEFORE LOOP)
*P
... (SELECTQ OBJPR & &) LOOP (FRPLACA DFPRP &) (FRPLACD DFPRP
&) (SELECTQ          user typed Control-E

*
```

Commands That Move Parentheses

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components thereof. Their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Of course, there will always be the same number of left parentheses as right parentheses in any list structure, since the parentheses are just a notational guide to the structure provided by PRINT. Thus, no command can insert or remove just one parenthesis, but this is suggestive of what actually happens.

In all six commands, *N* and *M* are used to specify an element of a list, usually of the current expression. In practice, *N* and *M* are usually positive or negative integers with the obvious interpretation. However, all six commands use the generalized NTH command (NTH *COM*) to find their element(s), so that *N*th element means the first element of the tail found by performing (NTH *N*). In other words, if the current expression is (LIST (CAR X) (SETQ Y (CONS W Z))), then (BI 2 CONS), (BI X -1), and (BI X Z) all specify the exact same operation.

All six commands generate an error if the element is not found, i.e., the NTH fails. All are undoable.

(BI *N M*)   [Editor Command]

"Both In". Inserts a left parentheses before the *N*th element and after the *M*th element in the current expression. Generates an error if the *M*th element is not contained in the *N*th tail, i.e., the *M*th element must be "to the right" of the *N*th element.

Example: If the current expression is (A B (C D E) F G), then (BI 2 4) will modify it to be (A (B (C D E) F) G).

(BI *N*)   [Editor Command]

Same as (BI *N N*).

Example: If the current expression is `(A B (C D E) F G)`, then `(BI -2)` will modify it to be `(A B (C D E) (F) G)`.

`(BO N)` [Editor Command]

"Both Out". Removes both parentheses from the _N_th element. Generates an error if _N_th element is not a list.

Example: If the current expression is `(A B (C D E) F G)`, then `(BO D)` will modify it to be `(A B C D E F G)`.

`(LI N)` [Editor Command]

"Left In". Inserts a left parenthesis before the _N_th element (and a matching right parenthesis at the end of the current expression), i.e. equivalent to `(BI N -1)`.

Example: if the current expression is `(A B (C D E) F G)`, then `(LI 2)` will modify it to be `(A (B (C D E) F G))`.

`(LO N)` [Editor Command]

"Left Out". Removes a left parenthesis from the _N_th element. *All elements following the _N_th element are deleted.* Generates an error if _N_th element is not a list.

Example: If the current expression is `(A B (C D E) F G)`, then `(LO 3)` will modify it to be `(A B C D E)`.

`(RI N M)` [Editor Command]

"Right In". Inserts a right parenthesis after the _M_th element of the _N_th element. The rest of the _N_th element is brought up to the level of the current expression.

Example: If the current expression is `(A (B C D E) F G)`, `(RI 2 2)` will modify it to be `(A (B C) D E F G)`. Another way of thinking about `RI` is to read it as "move the right parenthesis at the end of the _N_th element *in* to after its _N_th element."

`(RO N)` [Editor Command]

"Right Out". Removes the right parenthesis from the _N_th element, moving it to the end of the current expression. All elements following the _N_th element are moved inside of the _N_th element. Generates an error if _N_th element is not a list.

Example: If the current expression is (A B (C D E) F G), (RO 3) will modify it to be (A B (C D E F G)). Another way of thinking about RO is to read it as "move the right parenthesis at the end of the Nth element *out* to the end of the current expression."

TO and THRU

EXTRACT, EMBED, DELETE, REPLACE, and MOVE can be made to operate on several contiguous elements, i.e., a segment of a list, by using in their respective location specifications the TO or THRU command.

(@$_1$ THRU @$_2$)   [Editor Command]

Does a (LC . @$_1$), followed by an UP, and then a (BI 1 @$_2$), thereby grouping the segment into a single element, and finally does a 1, making the final current expression be that element.

For example, if the current expression is (A (B (C D) (E) (F G H) I) J K), following (C THRU G), the current expression will be ((C D) (E) (F G H)).

(@$_1$ TO @$_2$)   [Editor Command]

Same as THRU except the last element not included, i.e., after the BI, an (RI 1 -2) is performed.

If both @$_1$ and @$_2$ are numbers, and @$_2$ is greater than @$_1$, then @$_2$ counts from the beginning of the current expression, the same as @$_1$. In other words, if the current expression is (A B C D E F G), (3 THRU 5) means (C THRU E) not (C THRU G). In this case, the corresponding BI command is (BI 1 @$_2$-@$_1$+1).

THRU and TO are not very useful commands by themselves; they are intended to be used in conjunction with EXTRACT, EMBED, DELETE, REPLACE, and MOVE. After THRU and TO have operated, they set an internal editor flag informing the above commands that the element they are operating on is actually a segment, and that the extra pair of parentheses should be removed when the operation is complete. Thus:

```
*P
        (PROG (& & ATM IND VAL WORD) (PRIN1 & T) (PRIN1 & T) (SETQ
        IND &)
        (SETQ VAL &) **COMMENT** (SETQQ       user typed Control-E


*(MOVE (3 THRU 4) TO BEFORE 7)
        *P
        (PROG (& & ATM IND VAL WORD) (SETQ IND &) (SETQ VAL &)
        (PRIN1 & T)
        (PRIN1 & T) **COMMENT**       user typed Control-E
```

```
*
            *P
            (* FAIL RETURN FROM EDITOR. USER SHOULD NOTE THE VALUES OF
            SOURCEXPR
            AND CURRENTFORM.  CURRENTFORM IS THE LAST FORM IN SOURCEXPR
            WHICH WILL
            HAVE BEEN TRANSLATED, AND IT CAUSED THE ERROR.)
            *(DELETE (USER THRU CURR$))
            =CURRENTFORM.
            *P
            (* FAIL RETURN FROM EDITOR.CURRENTFORM IS       user typed Control-E



*
            *P
            ... LP (SELECTO & & & & NIL) (SETQ Y &) OUT (SETQ FLG &)
            (RETURN Y))
            *(MOVE (1 TO OUT) TO N HERE]
            *P
            ... OUT (SETQ FLG &) (RETURN Y) LP (SELECTQ & & & & NIL)
            (SETQ Y &))
            *

*PP
            [PROG (RF TEMP1 TEMP2)
               (COND
                  ((NOT (MEMB REMARG LISTING))
                      (SETQ TEMP1 (ASSOC REMARG NAMEDREMARKS))
            **COMMENT**
                      (SETQ TEMP2 (CADR TEMP1))
                      (GO SKIP))
                  (T    **COMMENT**
                      (SETQ TEMP1 REMARG)))
               (NCONC1 LISTING REMARG)
               (COND
                  ((NOT (SETQ TEMP2 (SASSOC


*(EXTRACT (SETQ THRU CADR) FROM COND)
            *P
            (PROG (RF TEMP1 TEMP2) (SETQ TEMP1 &) **COMMENT** (SETQ
            TEMP2 &) (NCONC1 LISTING REMARG) (COND & &    user typed Control-
            E

            *
```

TO and THRU can also be used directly with XTR, because XTR involves a location specification while
A, B, :, and MBD do not. Thus in the previous example, if the current expression had been the COND,
e.g.,you had first performed F COND, you could have used (XTR (SETQ THRU CADR)) to perform
the extraction.

$(@_1$ TO)   [Editor Command]

$(@_1$ THRU)   [Editor Command]

Both are the same as $(@_1$ THRU -1), i.e., from $@_1$ through the end of the list.

Examples:

```
*P
(VALUE (RPLACA DEPRP &) (RPLACD &) (RPLACA VARSWORD
&) (RETURN))
*(MOVE (2 TO) TO N (← PROG))
*(N (GO VAR))
*P
(VALUE (GO VAR))
*P
(T **COMMENT** (COND &) **COMMENT** (EDITSMASH CL &
&) (COND &))
*(-3 (GO REPLACE))
*(MOVE (COND TO) TO N ↑ PROG (N REPLACE))
*P
(T **COMMENT** (GO REPLACE))
*\ P
(PROG (&) **COMMENT** (COND & & &) (COND & & &)
DELETE (COND & &) REPLACE
(COND &) **COMMENT** (EDITSMASH CL & &) (COND &))
*

*PP
[LAMBDA (CLAUSALA X)
   (PROG (A D)
       (SETQ A CLAUSALA)
   LP  (COND
           ((NULL A)
               (RETURN)))
       (SERCH X A)
       (RUMARK (CDR A))
       (NOTICECL (CAR A))
       (SETQ A (CDR A))
       (GO LP]
*(EXTRACT (SERCH THRU NOT$) FROM PROG)
=NOTICECL
*P
(LAMBDA (CLAUSALA X) (SERCH X A) (RUMARK &) (NOTICECL
&))
*(EMBED (SERCH TO) IN (MAP CLAUSALA (FUNCTION (LAMBDA
(A) *]
*PP
[LAMBDA (CLAUSALA X)
   (MAP CLAUSALA
       (FUNCTION (LAMBDA (A)
               (SERCH X A)
               (RUMARK (CDR A))
               (NOTICECL (CAR A]
*
```

The R Command

`(R X Y)`   [Editor Command]

Replaces all instances of `X` by `Y` in the current expression, e.g., `(R  CAADR   CADAR)`.
Generates an error if there is not at least one instance.

The `R` command operates in conjunction with the search mechanism of the editor. The search
proceeds as described in the Search Algorithm section above, and `X` can employ any of the
patterns shown in the Commands That Search section above.   Each time `X` matches an
element of the structure, the element is replaced by (a copy of) `Y`; each time `X` matches a tail
of the structure, the tail is replaced by (a copy of) `Y`.

For example, if the current expression is `(A (B C) (B . C))`,

`(R C D)` will change it to `(A (B D) (B . D))`,

`(R (... . C) D)` will change it to `(A (B C) (B . D))`,

`(R C (D E))` will change it to `(A (B (D E)) (B D E))`, and

`(R (... . NIL) D)` will change it to `(A (B C . D) (B . C) . D)`.

If `X` is an atom or string containing `$`s (escapes), `$`s appearing in `Y` stand for the characters
matched by the corresponding `$` in `X`. For example, `(R FOO$ FIE$)` means for all atoms or
strings that begin with `FOO`, replace the characters "FOO" by "FIE". Applied to the list `(FOO
FOO2 XFOO1)`, *(R FOO$ FIE$)* would produce `(FIE FIE2 XFOO1)`, and `(R FOO
FIE)` would produce `(FIE FIE2 XFIE1)`. Similarly, `(R D A)` will change `(LIST
(CADR X) (CADDR Y))` to `(LIST (CAAR X) (CAADR))`. Note that `CADDR` was *not*
changed to `CAAAR`, i.e., `(R $D$ $A$)` does not mean replace every `D` with `A`, but replace the
first `D` in every atom or string by `A`. If you wanted to replace every `D` by `A`, you could perform
`(LP (R $D$ $A$))`.

You will be informed of all such `$` replacements by a message of the form `X->Y`, e.g., `CADR-
>CAAR`.

If `X` matches a string, it will be replaced by a string. It does not matter whether `X` or `Y`
themselves are strings, i.e. `(R $D$ $A$)`, `(R "$D$" $A$)`, `(R $D$ "$A$")`, and `(R
"D" "A")` are equivalent. `X` will never match with a number, i.e., `(R $1 $2)` will not
change 11 to 12.

The `$` (escape) feature can be used to delete or add characters, as well as replace them. For
example, `(R  $1  $)` will delete the terminating `1`'s from all literal atoms and strings.

Similarly, if an $ in *X* does not have a mate in *Y*, the characters matched by the $ are effectively deleted. For example, (R $/$ $) will change AND/OR to AND. There is no similar operation for changing AND/OR to OR, since the first $ in *Y* always corresponds to the first $ in *X*, the second **$** in *Y* to the second in *X*, etc. *Y* can also be a list containing $s, e.g., (R $1 (CAR $)) will change FOO1 to (CAR FOO), FIE1 to (CAR FIE).

If *X* does not contain $s, $ appearing in *Y* refers to the *entire* expression matched by *X*, e.g., (R LONGATOM '$) changes LONGATOM to 'LONGATOM, (R (SETQ X &) (PRINT $)) changes every (SETQ X &) to (PRINT (SETQ X &)). If *X* is a pattern containing an $ pattern somewhere *within* it, the characters matched by the $s are not available, and for the purposes of replacement, the effect is the same as though *X* did not contain any $s. For example, if you type (R (CAR F$) (PRINT *$*)), the second $ will refer to the entire expression matched by (CAR F$).

Since (R *X* *Y*) is a frequently used operation for **R**eplacing **C**haracters, the following command is provided:

(RC *X Y*)   [Editor Command]

Equivalent to (R *X* *Y*)

R and RC change all instances of *X* to *Y*. The commands R1 and RC1 are available for changing just one, (i.e., the first) instance of *X* to *Y*.

(R1 *X Y*)   [Editor Command]

Find the first instance of *X* and replace it by *Y*.

(RC1 *X Y*)   [Editor Command]

Equivalent to (R1 *X* *Y*).

In addition, while R and RC only operate within the current expression, R1 and RC1 will continue searching, a la the F command, until they find an instance of *x*, even if the search carries them beyond the current expression.

(SW *N M*)   [Editor Command]

Switches the *N*th and *M*th elements of the current expression.

For example, if the current expression is (LIST (CONS (CAR X) (CAR Y)) (CONS (CDR X) (CDR Y))), (SW 2 3) will modify it to be (LIST (CONS (CDR X) (CDR Y)) (CONS (CAR X) (CAR Y))). The relative order of *N* and *M* is not important, i.e., (SW 3 2) and (SW 2 3) are equivalent.

SW uses the generalized NTH command (NTH *COM*) to find the *N*th and *M*th elements, a la the BI-BO commands.

Thus in the previous example, (SW CAR CDR) would produce the same result.

(SWAP @$_1$ @$_2$)   [Editor Command]

Like SW except switches the expressions specified by @$_1$ and @$_2$, not the corresponding elements of the current expression, i.e. @$_1$ and @$_2$ can be at different levels in current expression, or one or both be outside of current expression.

Thus, using the previous example, (SWAP CAR CDR) would result in (LIST (CONS (CDR X) (CAR Y)) (CONS (CAR X) (CDR Y))).

Commands That Print

PP   [Editor Command]

Prettyprints the current expression.

P   [Editor Command]

Prints the current expression as though PRINTLEVEL (Chapter 25) were set to 2.

(P *M*)   [Editor Command]

Prints the *M*th element of the current expression as though PRINTLEVEL were set to 2.

(P *0*)   [Editor Command]

Same as P.

(P *M N*)   [Editor Command]

Prints the *M*th element of the current expression as though PRINTLEVEL were set to *N*.

(P 0 *N*)   [Editor Command]

Prints the current expression as though PRINTLEVEL were set to *N*.

?   [Editor Command]

Same as (P 0 100).

16-49

Both `(P M)` and `(P M N)` use the generalized `NTH` command `(NTH COM)` to obtain the corresponding element, so that *M* does not have to be a number, e.g., `(P COND  3)` will work. PP causes all comments to be printed as `**COMMENT**` (see Chapter 26). P and ? print as `**COMMENT**` only those comments that are (top level) elements of the current expression. Lower expressions are not really seen by the editor; the printing command simply sets `PRINTLEVEL` and calls `PRINT`.

`PP*`   [Editor Command]

   Prettyprints current expression, *including* comments.

   `PP*` is equivalent to `PP` except that it first resets `**COMMENT**FLG` to `NIL` (see Chapter 26).

`PPV`   [Editor Command]

   Prettyprints the current expression as a variable, i.e., no special treatment for `LAMBDA`, `COND`, `SETQ`, etc., or for CLISP.

`PPT`   [Editor Command]

   Prettyprints the current expression, printing CLISP translations, if any.

`?=`   [Editor Command]

   Prints the argument names and corresponding values for the current expression. Analagous to the `?=` break command (Chapter 14). For example,

```
*P
(STRPOS "A0???" X N (QUOTE ?) T)
*?=
X = "A0???"
Y = X
START = N
SKIP = (QUOTE ?)
ANCHOR = T
TAIL =
```

The command `MAKE` (see below) is an imperative form of `?=`. It allows you to specify a change to the element of the current expression that corresponds to a particular argument name.

All printing functions print to the terminal, regardless of the primary output file. All use the readtable `T`. No printing function ever changes the edit chain. All record the current edit chain for use by `\P` (above). All can be aborted with Control-E.

Commands for Leaving the Editor

OK   [Editor Command]

Exits from the editor.

STOP   [Editor Command]

Exits from the editor with an error. Mainly for use in conjunction with `TTY:` commands (see next section) that you want to abort.

Since all of the commands in the editor are errorset protected, you must exit from the editor via a command. `STOP` provides a way of distinguishing between a successful and unsuccessful (from your standpoint) editing session. For example, if you are executing `(MOVE 3 TO AFTER COND TTY:)`, and you exitsfrom the lower editor with an `OK`, the `MOVE` command will then complete its operation. If you want to abort the `MOVE` command, you must make the `TTY:` command generate an error. Do this by exiting from the lower editor with a `STOP` command. In this case, the higher editor's edit chain will not be changed by the `TTY:` command.

Actually, it is also possible to exit the editor by typing Control-D. `STOP` is preferred even if you are editing at the `EVALQT` level, as it will perform the necessary "wrapup" to insure that the changes made while editing will be undoable.

SAVE   [Editor Command]

Exits from the editor and saves the "state of the edit" on the property list of the function or variable being edited under the property `EDIT-SAVE`. If the editor is called again on the same structure, the editing is effectively "continued," i.e., the edit chain, mark list, value of `UNFIND` and `UNDOLST` are restored.

For example:

```
*P
(NULL X)
*F COND P
(COND (& &) (T &))
*SAVE
FOO
←   .
     .
     .
←EDITF(FOO)
EDIT
*P
(COND (& &) (T &))
*\ P
(NULL X)
*
```

SAVE is necessary only if you are editing many different expressions; an exit from the editor via OK always saves the state of the edit of that call to the editor on the property list of the atom EDIT, under the property name LASTVALUE. OK also remprops EDIT-SAVE from the property list of the function or variable being edited.

Whenever the editor is entered, it checks to see if it is editing the same expression as the last one edited. In this case, it restores the mark list and UNDOLST, and sets UNFIND to be the edit chain as of the previous exit from the editor. For example:

```
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
   .
   .
   .
*P
(COND & &)
*OK
FOO
←.
   .              any number of LISPX inputs
   .              except for calls to the editor
←EDITF(FOO)
EDIT
*P
(LAMBDA (X) (PROG & & LP & & & &))
*\ P
(COND & &)
*
```

Furthermore, as a result of the history feature, if the editor is called on the same expression within a certain number of LISPX inputs (namely, the size of the history list, which can be changed with CHANGESLICE, Chapter 13) the state of the edit of that expression is restored, regardless of how many other expressions may have been edited in the meantime. For example:

```
←EDITF(FOO)
EDIT
*
   .
   .
   .
*P
(COND (& &) (& &) (&) (T &))
*OK
FOO
   .              a small number of LISPX inputs,
   .              including editing
   .
←EDITF(FOO)
EDIT
*\ P
(COND (& &) (& &) (&) (T &))
*
```

Thus you can always continue editing, including undoing changes from a previous editing session, if one of the following occurs:

1. No other expressions have been edited since that session (since saving takes place at *exit* time, intervening calls that were aborted via Control-D or exited via STOP will not affect the editor's memory).

2. That session was "sufficiently" recent.

3. It was ended with a SAVE command.

Nested Calls to Editor

TTY: [Editor Command]

Calls the editor recursively. You can then type in commands, and have them executed. The TTY: command is completed when you exit from the lower editor (see OK and STOP above).

The TTY: command is extremely useful. It enables you to set up a complex operation, and perform interactive attention-changing commands part way through it. For example, the command (MOVE 3 TO AFTER COND 3 P TTY:) allows you to interact, in effect, *within* the MOVE command. You can then verify for yourself that the correct location has been found, or complete the specification "by hand." In effect, TTY: says "I'll tell you what you should do when you get there."

The TTY: command operates by printing TTY: and then calling the editor. The initial edit chain in the lower editor is the one that existed in the higher editor at the time the TTY: command was entered. Until you exit from the lower editor, any attention changing commands you execute only affect the lower editor's edit chain. Of course, if you perform any structure modification commands while under a TTY: command, these will modify the structure in both editors, since it is the same structure. When the TTY: command finishes, the lower editor's edit chain becomes the edit chain of the higher editor.

EF   [Editor Command]
EV   [Editor Command]
EP   [Editor Command]

Calls EDITF or EDITV or EDITP on CAR of current expression.

Manipulating the Characters of an Atom or String

RAISE   [Editor Command]

An edit macro defined as `UP` followed by `(I 1 (U-CASE (## 1)))`, i.e., it raises to uppercase the current expression, or if a tail, the first element of the current expression.

LOWER   [Editor Command]

Similar to `RAISE`, except uses `L-CASE`.

CAP   [Editor Command]

First does a `RAISE`, and then lowers all but the first character, i.e., the first character is left capitalized.

`RAISE`, `LOWER`, and `CAP` are all no-ops if the corresponding atom or string is already in that state.

(RAISE X)   [Editor Command]

Equivalent to `(I R (L-CASE X) X)`, i.e., changes every lowercase X to uppercase in the current expression.

(LOWER X)   [Editor Command]

Similar to `RAISE`, except performs `(I R X (L-CASE X))`.

In both `(RAISE X)` and `(LOWER X)`, X should be typed in uppercase.

REPACK   [Editor Command]

Permits the "editing" of an atom or string.

`REPACK` operates by calling the editor recursively on `UNPACK` of the current expression, or if it is a list, on `UNPACK` of its first element. If the lower editor is exited successfully, i.e., via `OK` as opposed to `STOP`, the list of atoms is made into a single atom or string, which replaces the atom or string being "repacked." The new atom or string is always printed.

Example:

```
*P
... "THIS IS A LOGN STRING")
*REPACK
*EDIT
P
(T H I S % I S % A % L O G N % S T R I N G)
```

```
*(SW G N)
*OK
"THIS IS A LONG STRING"
*
```

This could also have been accomplished by `(R $GN$ $NG$)` or simply `(RC GN NG)`.

`(REPACK @)` [Editor Command]

Does `(LC . @)` followed by `REPACK`, e.g. `(REPACK THIS$)`.

Manipulating Predicates and Conditional Expressions

`JOINC` [Editor Command]

Used to join two neighboring `COND`s together, e.g. $(\text{COND } CLAUSE_1 \ CLAUSE_2)$ followed by $(\text{COND } CLAUSE_3 \ CLAUSE_4)$ becomes $(\text{COND } CLAUSE_1 \ CLAUSE_2 \ CLAUSE_3 \ CLAUSE_4)$. `JOINC` does an `(F COND T)` first so that you don't have to be at the first `COND`.

`(SPLITC X)` [Editor Command]

Splits one `COND` into two. *X* specifies the last clause in the first `COND`, e.g. `(SPLITC 3)` splits $(\text{COND } CLAUSE_1 \ CLAUSE_2 \ CLAUSE_3 \ CLAUSE_4)$ into $(\text{COND } CLAUSE_1 \ CLAUSE_2)(\text{COND } CLAUSE_3 \ CLAUSE_4)$. Uses the generalized `NTH` command (`NTH COM`), so that *X* does not have to be a number, e.g., you can say `(SPLITC RETURN)`, meaning split after the clause containing `RETURN`. `SPLITC` also does an `(F COND T)` first.

`NEGATE` [Editor Command]

Negates the current expression, i.e. performs `(MBD  NOT)`, except that is smart about simplifying. For example, if the current expression is: `(OR  (NULL X) (LISTP  X))`, `NEGATE` would change it to `(AND X (NLISTP X))`.

`NEGATE` is implemented via the function `NEGATE` (Chapter 3).

`SWAPC` [Editor Command]

Takes a conditional expression of the form `(COND  (A B)(T  C))` and rearranges it to an equivalent `(COND ((NOT A) C)(T B))`, or `(COND (A B) (C D))` to `(COND ((NOT A) (COND (C D))) (T B))`.

SWAPC is smart about negations (uses NEGATE) and simplifying CONDs. It always produces an equivalent expression. It is useful for those cases where one wants to insert extra clauses or tests.

History Commands in the Editor

All of your inputs to the editor are stored on the history list EDITHISTORY (see Chapter 13, the editor's history list, and all of the programmer's assistant commands for manipulating the history list, e.g. REDO, USE, FIX, NAME, etc., are available for use on events on EDITHISTORY. In addition, the following four history commands are recognized specially by the editor. They always operate on the last, i.e. most recent, event.

DO *COM*   [Editor Command]

Allows you to supply the command name when it was omitted.

USE is useful when a command name is *incorrect*.

For example, suppose you want to perform (-2 (SETQ X (LIST Y Z))) but instead types just (SETQ X (LIST Y Z)). The editor will type SETQ ?, whereupon you can type DO -2. The effect is the same as though you had typed FIX, followed by (LI 1),(-1 -2), and OK, i.e., the command (-2 (SETQ X (LIST Y Z))) is executed. DO also works if the command is a line command.

!F   [Editor Command]

Same as DO F.

In the case of !F, the previous command is always treated as though it were a line command, e.g., if you type (SETQ X &) and then !F, the effect is the same as though you had typed F (SETQ X &), not (F (SETQ X &)).

!E   [Editor Command]

Same as DO E.

!N   [Editor Command]

Same as DO N.

NIL   [Editor Command]

Unless preceded by F or BF, is always a no-op. Thus extra right parentheses or square
brackets at the ends of commands are ignored.

CL   [Editor Command]

Clispifies the current expression (see Chapter 21).

DW   [Editor Command]

Dwimifies the current expression (see Chapter 21).

IFY   [Editor Command]

If the current statement is a COND statement (Chapter 9), replaces it with an eqivalent IF
statement.

GET*   [Editor Command]

If the current expression is a comment pointer (see Chapter 26), reads in the full text of the
comment, and replaces the current expression by it.

(* . *X*)   [Editor Command]

X is the text of a comment. * ascends the edit chain looking for a "safe" place to insert the
comment, e.g., in a COND clause, after a PROG statement, etc., and inserts (* . *X*) *after* that
point, if possible, otherwise before. For example, if the current expression is (FACT (SUB1
N)) in

```
        [COND
           ((ZEROP N) 1)
           (T (ITIMES N (FACT (SUB1 N]
```

then (* CALL FACT RECURSIVELY) would insert (* CALL FACT RECURSIVELY) *before*
the ITIMES expression. If inserted after the ITIMES, the comment would then be
(incorrectly) returned as the value of the COND. However, if the COND was itself a PROG
statement, and hence its value was not being used, the comment could be (and would be)
inserted after the ITIMES expression.

* does not change the edit chain, but UNFIND is set to where the comment was actually
inserted.

GETD   [Editor Command]

Essentially "expands" the current expression in line:

1. If (CAR of) the current expression is the name of a macro, expands the macro in line;

2. If a CLISP word, translates the current expression and replaces it with the translation;

3. If CAR is the name of a function for which the editor can obtain a symbolic definition, either in-core or from a file, substitutes the argument expressions for the corresponding argument names in the body of the definition and replaces the current expression with the result;

4. If CAR of the current expression is an open lambda, substitutes the arguments for the corresponding argument names in the body of the lambda, and then removes the lambda and argument list.

Warning: When expanding a function definition or open lambda expression, GETD does a simple substitution of the actual arguments for the formal arguments. Therefore, if any of the function arguments are used in other ways in the function definition (as functions, as record fields, etc.), they will simply be replaced with the actual arguments.

(MAKEFN (*FN . ACTUALARGS*) *ARGLIST N1 N2*)   [Editor Command]

The inverse of GETD: makes the current expression into a function. *FN* is the function name, *ARGLIST* its arguments. The argument names are substituted for the corresponding argument values in *ACTUALARGS*, and the result becomes the body of the function definition for *FN*. The current expression is then replaced with (*FN . ACTUALARGS*).

If N_1 and N_2 are supplied, (N_1 THRU N_2) is used rather than the current expression; if just N_1 is supplied, (N_1 THRU -1) is used.

If *ARGLIST* is omitted, MAKEFN will make up some arguments, using elements of *ACTUALARGS*, if they are literal atoms, otherwise arguments selected from (X Y Z A B C ...), avoiding duplicate argument names.

Example: If the current expression is (COND ((CAR X) (PRINT Y T)) (T (HELP))), then (MAKEFN (FOO (CAR X) Y) (A B)) will define FOO as (LAMBDA (A B) (COND (A (PRINT B T)) (T (HELP)))) and then replace the current expression with (FOO (CAR X) Y).

`(MAKE ARGNAME EXP)`   [Editor Command]

Makes the value of `ARGNAME` be `EXP` in the call which is the current expression, i.e. a `?=` command following a `MAKE` will always print `ARGNAME = EXP`. For example:

```
*P
(JSYS)
*?=
JSYS[N;AC1,AC2,AC3,RESULTAC]
*(MAKE N 10)
*(MAKE RESULTAC 3)
*P
(JSYS 10 NIL NIL NIL 3)
```

`Q`   [Editor Command]

Quotes the current expression, i.e. `MBD QUOTE`.

`D`   [Editor Command]

Deletes the current expression, then prints new current expression, i.e. `(:) I P`.

Commands That Evaluate

`E`   [Editor Command]

Causes the editor to call the Interlisp executive `LISPX` giving it the next input as argument. Example:

```
*E BREAK(FIE FUM)
(FIE FUM)
*E (FOO)

(FIE BROKEN)
:
```

`E` only works when when typed in, e.g, `(INSERT D BEFORE  E)` will treat `E` as a pattern, and search for `E`.

`(E X)`   [Editor Command]

Evaluates *X*, i.e., performs `(EVAL X)`, and prints the result on the terminal.

`(E X T)`   [Editor Command]

Same as `(E X)` but does not print.

The `(E X)` and `(E X T)` commands are mainly intended for use by macros and subroutine calls to the editor; you would probably type in a form for evaluation using the more convenient format of the (atomic) `E` command.

`(I C X`$_1$` ... X`$_N$`)`   [Editor Command]

Executes the *editor command* `(C Y`$_1$` ... Y`$_N$`)` where Y_i = `(EVAL X`$_i$`)`. If `C` is not an atom, `C` is evaluated also.

Examples:

`(I 3 (GETD 'FOO))` will replace the third element of the current expression with the definition of `FOO`.

`(I N FOO (CAR FIE))` will attach the value of `FOO` and `CAR` of the value of `FIE` to the end of the current expression.

`(I F = FOO T)` will search for an expression `EQ` to the value of `FOO`.

`(I (COND ((NULL FLG) '-1) (T 1)) FOO)`, if `FLG` is `NIL`, inserts the value of `FOO` before the first element of the current expression, otherwise replaces the first element by the value of `FOO`.

The `I` command sets an internal flag to indicate to the structure modification commands *not* to copy expression(s) when inserting, replacing, or attaching.

`EVAL`   [Editor Command]

Does an `EVAL` of the current expression.

`EVAL`, line-feed, and the `GO` command together effectively allows you to "single-step" a program through its symbolic definition.

`GETVAL`   [Editor Command]

Replaces the current expression by the result of evaluating it.

`(## COM`$_1$` COM`$_2$` ... COM`$_N$`)`   [NLambda NoSpread Function]

An nlambda, nospread function (not a command). Its value is what the current expression would be after executing the edit commands COM_1 ... COM_N starting from the present edit chain. Generates an error if any of COM_1 thru COM_N cause errors. The current edit chain is never changed.

Note: The `A`, `B`, `:`, `INSERT`, `REPLACE`, and `CHANGE` commands make special checks for `##` forms in the expressions used for inserting or replacing, and use a copy of `##` form instead (see the `A`,`B`, and `:` Commands section above). Thus, `(INSERT (## 3 2) AFTER 1)` is equivalent to `(I INSERT (COPY (## 3 2)) 'AFTER 1)`.

Example: `(I R 'X (## (CONS .. Z)))` replaces all `X`'s in the current expression by the first `CONS` containing a `Z`.

The `I` command is not very convenient for computing an *entire* edit command for execution, since it computes the command name and its arguments separately. Also, the `I` command cannot be used to compute an atomic command. The following two commands provide more general ways of computing commands.

`(COMS X`$_1$` ... X`$_M$`)`   [Editor Command]

Each `X`$_i$ is evaluated and its value is executed as a command.

For example, `(COMS (COND (X (LIST 1 X))))` will replace the first element of the current expression with the value of `X` if non-`NIL`, otherwise do nothing. The editor command `NIL` is a no-op (see the Miscellaneous Commands section above).

`(COMSQ COM`$_1$` ... COM`$_N$`)`   [Editor Command]

Executes `COM`$_1$` ... COM`$_N$.

`COMSQ` is mainly useful in conjunction with the `COMS` command. For example, suppose you want to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the `COMS` command. You would then write `(COMS (CONS 'COMSQ X))` where *X* computed the list of commands, e.g., `(COMS (CONS 'COMSQ (GETP FOO 'COMMANDS)))`.

Commands That Test

`(IF X)`   [Editor Command]

Generates an error *unless* the value of `(EVAL X)` is true. In other words, if `(EVAL X)` causes an error or `(EVAL X)` = `NIL`, `IF` will cause an error.

For some editor commands, the occurrence of an error has a well defined meaning, i.e., they use errors to branch on, as `COND` uses `NIL` and non-`NIL`. For example, an error condition in a location specification may simply mean "not this one, try the next." Thus the location

specification `(IPLUS (E (OR (NUMBERP (## 3))(ERROR!)) T))` specifies the first `IPLUS` whose second argument is a number. The `IF` command, by equating `NIL` to error, provides a more natural way of accomplishing the same result. Thus, an equivalent location specification is `(IPLUS (IF (NUMBERP (## 3))))`.

The `IF` command can also be used to select between two alternate lists of commands for execution.

`(IF X COMS`$_1$` COMS`$_2$`)` [Editor Command]

If `(EVAL X)` is true, execute $COMS_1$; if `(EVAL X)` causes an error or is equal to `NIL`, execute $COMS_2$.

Thus `IF` is equivalent to

```
(COMS (CONS 'COMSQ
         (COND
            ((CAR (NLSETQ (EVAL X)))
                COMS₁)
            (T   COMS₂))))
```

For example, the command `(IF (READP T) NIL (P))` will print the current expression provided the input buffer is empty.

`(IF X COMS`$_1$`)` [Editor Command]

If `(EVAL X)` is true, execute $COMS_1$; otherwise generate an error.

`(LP COMS`$_1$` ... COMS`$_N$`)` [Editor Command]

Repeatedly executes $COMS_1$ ... $COMS_N$ until an error occurs.

For example, `(LP F PRINT (N T))` will attach a `T` at the end of every `PRINT` expression. `(LP F PRINT (IF (## 3) NIL ((N T))))` will attach a `T` at the end of each print expression which does not already have a second argument. The form `(## 3)` will cause an error if the edit command `3` causes an error, thereby selecting `((N T))` as the list of commands to be executed. The `IF` could also be written as `(IF (CDDR (##)) NIL ((N T)))`.

When an error occurs, `LP` prints N `OCCURRENCES` where N is the number of times the commands were successfully executed. The edit chain is left as of the last complete successful execution of $COMS_1$ ... $COMS_N$.

(LPQ *COMS*$_1$ ... *COMS*$_N$) [Editor Command]

Same as LP but does not print the message *N* OCCURRENCES.

In order to prevent non-terminating loops, both LP and LPQ terminate when the number of iterations reaches MAXLOOP, initially set to 30. MAXLOOP can be set to NIL, which is equivalent to setting it to infinity. Since the edit chain is left as of the last successful completion of the loop, you can simply continue the LP command with REDO (see Chapter 13).

(SHOW *X*) [Editor Command]

X is a list of patterns. SHOW does a LPQ printing all instances of the indicated expression(s), e.g. (SHOW FOO (SETQ FIE &)) will print all FOOs and all (SETQ FIE &)s. Generates an error if there aren't any instances of the expression(s).

(EXAM *X*) [Editor Command]

Like SHOW except calls the editor recursively (via the TTY: command, see above) on each instance of the indicated espression(s) so that you can examine and/or change them.

(ORR *COMS*$_1$ ... *COMS*$_N$) [Editor Command]

ORR begins by executing *COMS*$_1$, a list of commands. If no error occurs, ORR is finished. Otherwise, ORR restores the edit chain to its original value, and continues by executing *COMS*$_2$, etc. If none of the command lists execute without errors, i.e., the ORR "drops off the end", ORR generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without an error.

NIL as a command list is perfectly legal, and will always execute successfully. Thus, making the last "argument" to ORR be NIL will insure that the ORR never causes an error. Any other atom is treated as (*ATOM*), i.e., the above example could be written as (ORR NX !NX NIL).

For example, (ORR (NX) (!NX) NIL) will perform a NX, if possible, otherwise a !NX, if possible, otherwise do nothing. Similarly, DELETE could be written as (ORR (UP (1)) (BK UP (2)) (UP (: NIL))).

Edit Macros

Many of the more sophisticated branching commands in the editor, such as ORR, IF, etc., are most often used in conjunction with edit macros. The macro feature permits you to define new commands and thereby expand the editor's repertoire, or redefine existing commands (to refer to the original

definition of a built-in command when redefining it via a macro, use the `ORIGINAL` command, below).

Macros are defined by using the `M` command:

$(M\ C\ COMS_1\ \ ...\ \ COMS_N)$ [Editor Command]

For C an atom, `M` defines C as an atomic command. If a macro is redefined, its new definition replaces its old. Executing C is then the same as executing the list of commands $COMS_1$ ... $COMS_N$.

For example, `(M BP BK UP P)` will define `BP` as an atomic command which does three things, a `BK`, and `UP`, and a `P`. Macros can use commands defined by macros as well as built in commands in their definitions. For example, suppose `Z` is defined by `(M Z -1 (IF (READP T) NIL (P)))`, i.e., `Z` does a `-1`, and then if nothing has been typed, a `P`. Now we can define `ZZ` by `(M ZZ -1 Z)`, and `ZZZ` by `(M ZZZ -1 -1 Z)` or `(M ZZZ -1 ZZ)`.

Macros can also define list commands, i.e., commands that take arguments.

$(M\ (C)\ (ARG_1\ \ ...\ \ ARG_N)\ COMS_1\ \ ...\ \ COMS_M)$ [Editor Command]

C an atom. `M` defines C as a list command. Executing $(C\ E_1\ \ ...\ \ E_N)$ is then performed by substituting E_1 for ARG_1, ... E_N for ARG_N throughout $COMS_1$ ... $COMS_M$, and then executing $COMS_1$ ... $COMS_M$.

For example, we could define a more general `BP` by `(M (BP) (N) (BK N) UP P)`. Thus, `(BP 3)` would perform `(BK 3)`, followed by an `UP`, followed by a `P`.

A list command can be defined via a macro so as to take a fixed or indefinite number of "arguments", as with spread vs. nospread functions. The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the "argument list" is *atomic*, the command takes an indefinite number of arguments.

$(M\ (C)\ ARG\ COMS_1\ \ ...\ \ COMS_M)$ [Editor Command]

If C, ARG are both atoms, this defines C as a list command. Executing $(C\ E_1\ \ ...\ \ E_N)$ is performed by substituting $(E_1\ \ ...\ \ E_N)$, i.e., `CDR` of the command, for ARG throughout $COMS_1$ ... $COMS_M$, and then executing $COMS_1$ ... $COMS_M$.

For example, the command `2ND` (see the Location Specification section above), could be defined as a macro by `(M (2ND) X (ORR ((LC . X) (LC . X))))`.

For all editor commands, "built in" commands as well as commands defined by macros as atomic commands and list definitions are *completely* independent. In other words, the existence of an atomic definition for `C` in *no* way affects the treatment of `C` when it appears as `CAR` of a list command, and the existence of a list definition for `C` in *no* way affects the treatment of `C` when it appears as an atom. In particular, `C` can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used.

Once `C` is defined as an atomic command via a macro definition, it will *not* be searched for when used in a location specification, unless it is preceded by an `F`. Thus `(INSERT -- BEFORE BP)` would not search for `BP`, but instead perform a `BK`, and `UP`, and a `P`, and then do the insertion. The corresponding also holds true for list commands.

Occasionally, your will want to employ the `S` command in a macro to save some temporary result. For example, the `SW` command could be defined as:

```
(M (SW) (N M)
    (NTH N)
    (S FOO 1)
    MARK
    0
    (NTH M)
    (S FIE 1)
    (I 1 FOO)
    ←←
    (I 1 FIE))
```

Since this version of `SW` sets `FOO` and `FIE`, using `SW` may have undesirable side effects, especially when the editor was called from deep in a computation, we would have to be careful to make up unique names for dummy variables used in edit macros, which is bothersome. Furthermore, it would be impossible to define a command that called itself recursively while setting free variables. The `BIND` command solves both problems.

`(BIND `$COMS_1$` ... `$COMS_N$`)`   [Editor Command]

Binds three dummy variables `#1`, `#2`, `#3`, (initialized to `NIL`), and then executes the edit commands $COMS_1$ ... $COMS_N$. `BIND` uses a `PROG` to make these bindings, so they are only in effect while the commands are being executed and `BIND`s can be used recursively; the variables `#1`, `#2`, and `#3` will be rebound each time `BIND` is invoked.

Thus, we can write SW safely as:

```
(M (SW) (N M)
    (BIND (NTH N)
          (S #1 1)
          MARK
          0
          (NTH M)
          (S #2 1)
          (I 1 #1)
```

$$\leftarrow\leftarrow \qquad (\text{I } 1 \ \#2)))$$

(ORIGINAL $COMS_1$ ... $COMS_N$)   [Editor Command]

Executes $COMS_1$ ... $COMS_N$ without regard to macro definitions. Useful for redefining a built in command in terms of itself., i.e. effectively allows you to "advise" edit commands.

User macros are stored on a list USERMACROS. The file package command USERMACROS (Chapter 17) is available for dumping all or selected user macros.

Undo

Each command that causes structure modification automatically adds an entry to the front of UNDOLST that contains the information required to restore all pointers that were changed by that command.

UNDO   [Editor Command]

Undoes the last, i.e., most recent, structure modification command that has not yet been undone, and prints the name of that command, e.g., MBD undone. The edit chain is then *exactly* what it was before the "undone" command had been performed. If there are no commands to undo, UNDO types nothing saved.

!UNDO   [Editor Command]

Undoes all modifications performed during this editing session, i.e. this call to the editor. As each command is undone, its name is printed a la UNDO. If there is nothing to be undone, !UNDO prints nothing saved.

Undoing an event containing an I, E, or S command will also undo the side effects of the evaluation(s), e.g., undoing (I 3 (/NCONC FOO FIE)) will not only restore the third element but also restore FOO. Similarly, undoing an S command will undo the set. See the discussion of UNDO in Chapter 13. (If the I command was typed directly to the editor, /NCONC would automatically be substituted for NCONC as described in Chapter 13.)

Since UNDO and !UNDO cause structure modification, they also add an entry to UNDOLST. However, UNDO and !UNDO entries are skipped by UNDO, e.g., if you perform an INSERT, and then an MBD, the first UNDO will undo the MBD, and the second will undo the INSERT. However, you can also specify precisely which commands you want undone by identifying the corresponding entry. In this case, you can undo an UNDO command, e.g., by typing UNDO UNDO, or undo a !UNDO command, or undo a command other than that most recently performed.

Whenever you *continue* an editing session, the undo information of the previous session is protected by inserting a special blip, called an undo-block, on the front of UNDOLST. This undo-block will terminate the operation of a !UNDO, thereby confining its effect to the current session, and will similarly prevent an UNDO command from operating on commands executed in the previous session.

Thus, if you enter the editor continuing a session, and immediately execute an UNDO or !UNDO, the editor will type BLOCKED instead of NOTHING SAVED. Similarly, if you execute several commands and then undo them all, another UNDO or !UNDO will also cause BLOCKED to be typed.

    UNBLOCK   [Editor Command]

      Removes an undo-block. If executed at a non-blocked state, i.e., if UNDO or !UNDO *could* operate, types NOT BLOCKED.

    TEST   [Editor Command]

      Adds an undo-block at the front of UNDOLST.

      Note that TEST together with !UNDO provide a "tentative" mode for editing, i.e., you can perform a number of changes, and then undo all of them with a single !UNDO command.

    (UNDO *EventSpec*)   [Editor Command]

      EventSpec is an event specification (see Chapter 13). Undoes the indicated event on the history list. In this case, the event does not have to be in the current editing session, even if the previous session has not been unblocked as described above. However, you do have to be editing the same expression as was being edited in the indicated event.

      If the expressions differ, the editor types the warning message "different expression," and does not undo the event. The editor enforces this to avoid your accidentally undoing a random command by giving the wrong event specification.

EDITDEFAULT

Whenever a command is not recognized, i.e., is not "built in" or defined as a macro, the editor calls an internal function, EDITDEFAULT, to determine what action to take. Since EDITDEFAULT is part of the edit block, you cannot advise or redefine it as a means of augmenting or extending the editor. However, you can accomplish this via EDITUSERFN. If the value of the variable EDITUSERFN is T, EDITDEFAULT calls the function EDITUSERFN giving it the command as an argument. If EDITUSERFN returns a non-NIL value, its value is interpreted as a single command and executed. Otherwise, the error correction procedure described below is performed.

If a location specification is being executed, an internal flag informs EDITDEFAULT to treat the command as though it had been preceded by an F.

If the command is a list, an attempt is made to perform spelling correction on the CAR of the command (unless DWIMFLG = NIL) using EDITCOMSL, a list of all list edit commands. If spelling correction is successful, the correct command name is RPLACAed into the command, and the editor continues by executing the command. In other words, if you type (LP F PRINT (MBBD AND (NULL FLG))), only one spelling correction will be necessary to change MBBD to MBD. If spelling correction is not successful, an error is generated.

> Note: When a macro is defined via the M command, the command name is added to
> EDITCOMSA or EDITCOMSL, depending on whether it is an atomic or list
> command.   The USERMACROS file package command is aware of this, and
> provides for restoring EDITCOMSA and EDITCOMSL.

If the command is atomic, the procedure followed is a little more elaborate.

1.   If the command is one of the list commands, i.e., a member of EDITCOMSL, and there is additional input on the same terminal line, treat the entire line as a single list command. The line is read using READLINE (see Chapter 13), so the line can be terminated by a square bracket, or by a carriage return not preceded by a space. You may omit parentheses for any list command typed in at the top level (provided the command is not also an atomic command, e.g. NX, BK). For example,

```
 *P
        (COND (& &) (T &))
        *XTR 3 2]
        *MOVE TO AFTER LP
        *
```

If the command is on the list EDITCOMSL but no additional input is on the terminal line, an error is generated. For example:

```
 *P
        (COND (& &) (T &))
        *MOVE

        MOVE ?
        *
```

If the command is on EDITCOMSL, and *not* typed in directly, e.g., it appears as one of the commands in a LP command, the procedure is similar, with the rest of the command stream at that level being treated as "the terminal line", e.g. (LP F (COND (T &)) XTR 2 2).

If the command is being executed in location context, EDITDEFAULT does not get this far, e.g., (MOVE TO AFTER COND XTR 3) will search for XTR, *not* execute it. However, (MOVE TO AFTER COND (XTR 3)) will work.

2.	If the command was typed in and the first character in the command is an 8, treat the 8 as a mistyped left parenthesis, and and the rest of the line as the arguments to the command, e.g.,

```
*P
        (COND (& &) (T &))
        *8-2 (Y (RETURN Z)))
        =(-2
        *P
        (COND (Y &) (& &) (T &))
```

3.	If the command was typed in, is the name of a function, and is followed by NIL or a list CAR of which is not an edit command, assume you forgot to type E and intend to apply the function to its arguments, type =E and the function name, and perform the indicated computation, e.g.

```
*BREAK(FOO)
        =E BREAK
        (FOO)
        *
```

4.	If the last character in the command is P, and the first *N*-1 characters comprise a number, assume that you intended two commands, e.g.,

```
*P
        (COND (& &) (T &))
        *0P
        =0 P
        (SETQ X (COND & &))
```

5.	Attempt spelling correction using EDITCOMSA, and if successful, execute the corrected command.

6.	If there is additional input on the same line, or command stream, spelling correct using EDITCOMSL as a spelling list, e.g.,

```
*MBBD SETQ X
        =MBD
        *
```

7.	Otherwise, generate an error.

Time Stamps

Whenever a function is edited, and changes were made, the function is time-stamped (by EDITE), which consists of inserting a comment of the form (* *USERS-INITIALS DATE*). *USERS-INITIALS* is the value of the variable INITIALS. After greeting (see Chapter 12), the function SETINITIALS is

called. `SETINITIALS` searches `INITIALSLST`, a list of elements of the form (`USERNAME` . `INITIALS`) or (`USERNAME FIRSTNAME INITIALS`). If your name is found, `INITIALS` is set accordingly. If your username name is *not* found on `INITIALSLST`, `INITIALS` is set to the value of `DEFAULTINITIALS`, initially `edited:`. Thus, the default is to always time stamp. To suppress time stamping, you must either include an entry of the form (`USERNAME`) on `INITIALSLST`, or set `DEFAULTINITIALS` to `NIL` before greeting, i.e. in your user profile, or else, *after* greeting, explicitly set `INITIALS` to `NIL`.

If you want your functions to be time stamped with your initials when edited, include a file package command command of the form (`ADDVARS (INITIALSLST(USERNAME . INITIALS)))` in your `INIT.LISP` file (see Chapter 12).

The following three functions may be of use for specialized applications with respect to time-stamping: (`FIXEDITDATE EXPR`) which, given a lambda expression, inserts or smashes a time-stamp comment; (`EDITDATE? COMMENT`) which returns `T` if `COMMENT` is a time stamp; and (`EDITDATE OLDATE INITLS`) which returns a new time-stamp comment. If `OLDATE` is a time-stamp comment, it will be reused.

Warning with Declarations

CAUTION: There is a feature of the BYTECOMPILER that is not supported by SEdit or the XCL compiler. It is possible to insert a comment at the beginning of your function that looks like

(* DECLARATIONS: --)

The tail, or -- section, of this comment is taken as a set of local record declarations which are then used by the compiler in that function just as if they had been declared globally. See the "Compiler" section in Chapter 3 of these Notes for additional behavior in XCL.

SEdit does not recognize such declarations. Thus, if the "Expand" command is used, the expansion will not be done with these record declarations in effect. The code that you see in SEdit will not be the same code compiled by the BYTECOMPILER.

[This page intentionally left blank]

17. FILE MANAGER

*Warning: The subsystem within Medley used for managing collections of definitions (of functions, variables, etc.) is known as the "File Manager." This terminology is confusing, because the word "file" is also used in the more conventional sense as meaning a collection of data stored on some physical media. Unfortunately, it is not possible to change this terminology at this time, because many functions and variables (*MAKEFILE, FILEPKGTYPES*, etc.) incorporate the word "file" in their names.*

Most implementations of Lisp treat symbolic files as unstructured text, much as they are treated in most conventional programming environments. Function definitions are edited with a character-oriented text editor, and then the changed definitions (or sometimes the entire file) is read or compiled to install those changes in the running memory image. Interlisp incorporates a different philosophy. A symbolic file is considered as a database of information about a group of data objects---function definitions, variable values, record declarations, etc. The text in a symbolic file is never edited directly. Definitions are edited only after their textual representations on files have been converted to data-structures that reside inside the Lisp address space. The programs for editing definitions inside Medley can therefore make use of the full set of data-manipulation capabilities that the environment already provides, and editing operations can be easily intermixed with the processes of evaluation and compilation.

Medley is thus a "resident" programming environment, and as such it provides facilities for moving definitions back and forth between memory and the external databases on symbolic files, and for doing the bookkeeping involved when definitions on many symbolic files with compiled counterparts are being manipulated. The file manager provides those capabilities. It shoulders the burden of keeping track of where things are and what things have changed so that you don't have to. The file manager also keeps track of which files have been modified and need to be updated and recompiled.

The file manager is integrated into many other system packages. For example, if only the compiled version of a file is loaded and you attempt to edit a function, the file manager will attempt to load the source of that function from the appropriate symbolic file. In many cases, if a datum is needed by some program, the file manager will automatically retrieve it from a file if it is not already in your working environment.

Some of the operations of the file manager are rather complex. For example, the same function may appear in several different files, or the symbolic or compiled files may be in different directories, etc. Therefore, this chapter does not document how the file manager works in each and every situation, but instead makes the deliberately vague statement that it does the "right" thing with respect to keeping track of what has been changed, and what file operations need to be performed in accordance with those changes.

For a simple illustration of what the file manager does, suppose that the symbolic file FOO contains the functions FOO1 and FOO2, and that the file BAR contains the functions BAR1 and BAR2. These two files could be loaded into the environment with the function LOAD:

```
← (LOAD 'FOO)
FILE CREATED  4-MAR-83 09:26:55
```

```
                    FOOCOMS
                    {DSK}FOO.;1

                    ← (LOAD 'BAR)
                    FILE CREATED  4-MAR-83 09:27:24
                    BARCOMS
                    {DSK}BAR.;1
```

Now, suppose that we change the definition of FOO2 with the editor, and we define two new functions, NEW1 and NEW2. At that point, the file manager knows that the in-memory definition of FOO2 is no longer consistent with the definition in the file FOO, and that the new functions have been defined but have not yet been associated with a symbolic file and saved on permanent storage. The function FILES? summarizes this state of affairs and enters into an interactive dialog in which we can specify what files the new functions are to belong to.

```
                    ← (FILES?)
                    FOO...to be dumped.
                        plus the functions: NEW1,NEW2
                    want to say where the above go ? Yes
                    (functions)
                    NEW1  File name: BAR
                    NEW2  File name: ZAP
                      new file ? Yes
                    NIL
```

The file manager knows that the file FOO has been changed, and needs to be dumped back to permanent storage. This can be done with MAKEFILE.

```
                    ←(MAKEFILE 'FOO)
                    {DSK}FOO.;2
```

Since we added NEW1 to the old file BAR and established a new file ZAP to contain NEW2, both BAR and ZAP now also need to be dumped. This is confirmed by a second call to FILES?:

```
                    ←(FILES?)
                    BAR, ZAP...to be dumped.
                    FOO...to be listed.
                    FOO...to be compiled
                    NIL
```

We are also informed that the new version we made of FOO needs to be listed (sent to a printer) and that the functions on the file must be compiled.

Rather than doing several MAKEFILEs to dump the files BAR and ZAP, we can simply call CLEANUP. Without any further user interaction, this will dump any files whose definitions have been modified. CLEANUP will also send any unlisted files to the printer and recompile any files which need to be recompiled. CLEANUP is a useful function to use at the end of a debugging session. It will call FILES? if any new objects have been defined, so you do not lose the opportunity to say explicitly where those belong. In effect, the function CLEANUP executes all the operations necessary to make the your permanent files consistent with the definitions in the current core-image.

```
                    ← (CLEANUP)
                    FOO...compiling {DSK}FOO.;2
```

```
                    .
                    .
                    .
        BAR...compiling {DSK}BAR.;2
                    .
                    .
                    .
        ZAP...compiling {DSK}ZAP.;1
                    .
                    .
                    .
```

In addition to the definitions of functions, symbolic files in Interlisp can contain definitions of a variety of other types, e.g. variable values, property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file manager uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a symbol), a definition of a given type (called the file manager type). Note that the same name may have several definitions of different types. For example, a symbol may have both a function definition and a variable definition. The file manager also keeps track of the files that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

Symbolic files on permanent storage devices are referred to by names that obey the naming conventions of those devices, usually including host, directory, and version fields. When such definition groups are noticed by the file manager, they are assigned simple *root names* and these are used by all file manager operations to refer to those groups of definitions. The root name for a group is computed from its full permanent storage name by applying the function ROOTFILENAME; this strips off the host, directory, version, etc., and returns just the simple name field of the file. For each file, the file manager also has a data structure that describes what definitions it contains. This is known as the commands of the file, or its "filecoms". By convention, the filecoms of a file whose root name is X is stored as the value of the symbol XCOMS. For example, the value of FOOCOMS is the filecoms for the file FOO. This variable can be directly manipulated, but the file manager contains facilities such as FILES? which make constructing and updating filecoms easier, and in some cases automatic. See the Functions for Manipulating File Command Lists section.

The file manager is able to maintain its databases of information because it is notified by various other routines in the system when events take place that may change that database. A file is "noticed" when it is loaded, or when a new file is stored (though there are ways to explicitly notice files without completely loading all their definitions). Once a file is noticed, the file manager takes it into account when modifying filecoms, dumping files, etc. The file manager also needs to know what typed definitions have been changed or what new definitions have been introduced, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file manager operations (LOAD, TCOMPL, PRETTYDEF, etc.), as well as those functions that define or change data, (EDITF, EDITV, EDITP, DWIM corrections to user functions) interact with the file manager. Also, *typed-in* assignment of variables or property values is noticed by the file manager. (Note that modifications to variable or property values during the execution of a function body are not noticed.) In some cases the marking procedure can be subtle, e.g. if you edit a property list using EDITP, only those properties whose values are actually changed (or added) are marked.

All file manager operations can be disabled with `FILEPKGFLG`.

FILEPKGFLG   [Variable]

> The file manager can be disabled by setting `FILEPKGFLG` to `NIL`. This will turn off noticing files and marking changes. `FILEPKGFLG` is initially `T`.

The rest of this chapter goes into further detail about the file manager. Functions for loading and storing symbolic files are presented first, followed by functions for adding and removing typed definitions from files, moving typed definitions from one file to another, determining which file a particular definition is stored in, and so on.

Loading Files

The functions below load information from symbolic files into the Interlisp environment. A symbolic file contains a sequence of Interlisp expressions that can be evaluated to establish specified typed definitions. The expressions on symbolic files are read using `FILERDTBL` as the read table.

The loading functions all have an argument *LDFLG*. *LDFLG* affects the operation of `DEFINE`, `DEFINEQ`, `RPAQ`, `RPAQ?`, and `RPAQQ`. While a source file is being loaded, `DFNFLG` (Chapter 10) is rebound to *LDFLG*. Thus, if *LDFLG* = `NIL`, and a function is redefined, a message is printed and the old definition saved. If *LDFLG* = `T` , the old definition is simply overwritten. If *LDFLG* = `PROP`, the functions are stored as "saved" definitions on the property lists under the property `EXPR` instead of being installed as the active definitions. If *LDFLG* = `ALLPROP`, not only function definitions but also variables set by `RPAQQ`, `RPAQ`, `RPAQ?` are stored on property lists (except when the variable has the value `NOBIND`, in which case they are set to the indicated value regardless of `DFNFLG`).

Another option is available for loading systems for others to use and who wish to suppress the saving of information used to aid in development and debugging. If *LDFLG* = `SYSLOAD`, `LOAD` will:

1.   Rebind DFNFLG to T, so old definitions are simply overwritten

2.   Rebind LISPXHIST to NIL, thereby making the LOAD not be undoable and eliminating the cost of saving undo information (Chapter 13)

3.   Rebind ADDSPELLFLG to NIL, to suppress adding to spelling lists

4.   Rebind FILEPKGFLG to NIL, to prevent the file from being "noticed" by the file manager

5.   Rebind BUILDMAPFLG to NIL, to prevent a file map from being constructed

6.   After the load has completed, set the filecoms variable and any filevars variables to NOBIND

7.   Add the file name to SYSFILES rather than FILELST

A filevars variable is any variable appearing in a file manager command of the form (*FILECOM* * *VARIABLE*) (see the FileVars section). Therefore, if the filecoms includes (FNS * FOOFNS), FOOFNS is set to NOBIND. If you want the value of such a variable to be retained, even when the file is loaded with *LDFLG* = SYSLOAD, then you should replace the variable with an equivalent, *non-atomic* expression, such as (FNS * (PROGN FOOFNS)).

All functions that have *LDFLG* as an argument perform spelling correction using LOADOPTIONS as a spelling list when *LDFLG* is not a member of LOADOPTIONS. LOADOPTIONS is initially (NIL T PROP ALLPROP SYSLOAD).

(**LOAD** *FILE LDFLG PRINTFLG*)   [Function]

>   Reads successive expressions from *FILE* (with FILERDTBL as read table) and evaluates each as it is read, until it reads either NIL, or the single atom STOP. Note that LOAD can be used to load both symbolic and compiled files. Returns *FILE* (full name).

>   If *PRINTFLG* = T, LOAD prints the value of each expression; otherwise it does not.

(**LOAD?** *FILE LDFLG PRINTFLG*)   [Function]

>   Similar to LOAD except that it does not load *FILE* if it has already been loaded, in which case it returns NIL.

>   LOAD? loads *FILE* except when the *same* version of the file has been loaded (either from the same place, or from a copy of it from a different place). Specifically, LOAD? considers that *FILE* has already been loaded if the full name of *FILE* is on LOADEDFILELST (see the Noticing Files section) or the date stored on the FILEDATES property of the root file name of *FILE* is the same as the FILECREATED expression on *FILE*.

(**LOADFNS** *FNS FILE LDFLG VARS*)   [Function]

>   Permits selective loading of definitions. *FNS* is a list of function names, a single function name, or T, meaning to load all of the functions on the file. *FILE* can be either a compiled

or symbolic file. If a compiled definition is loaded, so are all compiler-generated subfunctions. The interpretation of *LDFLG* is the same as for `LOAD`.

If *FILE* = `NIL`, `LOADFNS` will use `WHEREIS` (see the Storing Files section) to determine where the first function in *FNS* resides, and load from that file. Note that the file must previously have been "noticed". If `WHEREIS` returns `NIL`, and the `WHEREIS` library package has been loaded, `LOADFNS` will use the `WHEREIS` data base to find the file containing *FN*.

VARS specifies which non-`DEFINEQ` expressions are to be loaded (i.e., evaluated). It is interpreted as follows:

T	Means to load all non-`DEFINEQ` expressions.
NIL	Means to load none of the non-`DEFINEQ` expressions.
VARS	Means to evaluate all variable assignment expressions (beginning with `RPAQ`, `RPAQQ`, or `RPAQ?`, see the Functions Used Within Source Files section).
Any other symbol	Means the same as specifying a list containing that atom.
A list	If *VARS* is a list that is not a valid function definition, each element in *VARS* is "matched" against each non-`DEFINEQ` expression, and if any elements in *VARS* "match" successfully, the expression is evaluated. "Matching" is defined as follows: If an element of *VARS* is an atom, it matches an expression if it is `EQ` to either the `CAR` or the `CADR` of the expression. If an element of *VARS* is a list, it is treated as an edit pattern (see Chapter 16), and matched with the entire expression (using `EDIT4E`, described in Chapter 16). For example, if *VARS* was `(FOOCOMS DECLARE: (DEFLIST & (QUOTE MACRO)))`, this would cause `(RPAQQ FOOCOMS ...)`, all `DECLARE:`s, and all `DEFLIST`s which set up `MACRO`s to be read and evaluated.
A function definition	If *VARS* is a list and a valid function definition (`(FNTYP VARS)` is true), then `LOADFNS` will invoke that function on every non-`DEFINEQ` expression being considered, applying it to two arguments, the first and second elements in the expression. If the function returns `NIL`, the expression will be skipped; if it returns a non-`NIL` symbol (e.g., `T`), the expression will be evaluated; and if it returns a list, this list is evaluated instead of the expression. The file pointer is set to the very beginning of the expression before calling the *VARS* function definition, so it may read the entire expression if necessary. If the function returns a symbol, the file pointer is reset and the expression is `READ` or `SKREAD`. However, the file pointer is not reset when the function returns a list, so the

function must leave it set immediately after the expression
that it has presumably read.

LOADFNS returns a list of:

1. The names of the functions that were found

2. A list of those functions not found (if any) headed by the symbol NOT-
   FOUND:

3. All of the expressions that were evaluated

4. A list of those members of *VARS* for which no corresponding expressions were
   found (if any), again headed by the symbol NOT-FOUND:

For example:

```
← (LOADFNS '(FOO FIE FUM) FILE NIL '(BAZ (DEFLIST &)))
(FOO FIE (NOT-FOUND: FUM) (RPAQ BAZ ...) (NOT-FOUND:
(DEFLIST &)))
```

(**LOADVARS** *VARS FILE LDFLG*) [Function]

    Same as (LOADFNS NIL *FILE LDFLG VARS*).

(**LOADFROM** *FILE FNS LDFLG*) [Function]

    Same as (LOADFNS *FNS FILE LDFLG* T).

Once the file manager has noticed a file, you can edit functions contained in the file without explicitly
loading them. Similarly, those functions which have not been modified do not have to be loaded in
order to write out an updated version of the file. Files are normally noticed (i.e., their contents
become known to the file manager) when either the symbolic or compiled versions of the file are
loaded. If the file is *not* going to be loaded completely, the preferred way to notice it is with
LOADFROM. You can also load some functions at the same time by giving LOADFROM a second
argument, but it is normally used simply to inform the file manager about the existence and contents
of a particular file.

(**LOADBLOCK** *FN FILE LDFLG*) [Function]

    Calls LOADFNS on those functions contained in the block declaration containing *FN* (see
    Chapter 18). LOADBLOCK is designed primarily for use with symbolic files, to load the
    EXPRs for a given block. It will not load a function which already has an in-core EXPR
    definition, and it will not load the block name, unless it is also one of the block functions.

(**LOADCOMP** *FILE LDFLG*) [Function]

    Performs all operations on *FILE* associated with compilation, i.e. evaluates all expressions
    under a DECLARE: EVAL@COMPILE, and "notices" the function and variable names by
    adding them to the lists NOFIXFNSLST and NOFIXVARSLST (see Chapter 21).

Thus, if building a system composed of many files with compilation information scattered among them, all that is required to compile one file is to LOADCOMP the others.

(**LOADCOMP?** *FILE LDFLG*)   [Function]

Similar to LOADCOMP, except it does not load if file has already been loaded (with LOADCOMP), in which case its value is NIL.

LOADCOMP? will load the file even if it has been loaded with LOAD, LOADFNS, etc. The only time it will not load the file is if the file has already been loaded with LOADCOMP.

FILESLOAD provides an easy way for you to load a series of files, setting various options:

(**FILESLOAD** *FILE$_1$ ... FILE$_N$*)   [NLambda NoSpread Function]

Loads the files *FILE$_1$ ... FILE$_N$* (all arguments unevaluated). If any of these arguments are lists, they specify certain loading options for all following files (unless changed by another list). Within these lists, the following commands are recognized:

FROM *DIR*   Search the specified directories for the file. *DIR* can either be a single directory, or a list of directories to search in order. For example, (FILESLOAD (FROM {ERIS}<LISPCORE>SOURCES>) ...) will search the directory {ERIS}<LISPCORE>SOURCES> for the files. If this is not specified, the default is to search the contents of DIRECTORIES (see Chapter 24).

If FROM is followed by the key word VALUEOF, the following word is evaluated, and the value is used as the list of directories to search. For example, (FILESLOAD (FROM VALUEOF FOO) ...) will search the directory list that is the value of the variable FOO.

As a special case, if *DIR* is a symbol, and the symbol *DIR*DIRECTORIES is bound, the value of this variable is used as the directory search list. For example, since the variable LISPUSERSDIRECTORIES (see Chapter 24) is commonly used to contain a list of directories containing "library" packages, (FILESLOAD (FROM LISPUSERS) ...) can be used instead of (FILESLOAD (FROM VALUEOF LISPUSERSDIRECTORIES) ...)

If a FILESLOAD is read and evaluated while loading a file, and it doesn't contain a FROM expression, the default is to search the directory containing the FILESLOAD expression before the value of DIRECTORIES. FILESLOAD expressions can be dumped on files using the FILES file manager command.

SOURCE	Load the source version of the file rather than the compiled version.
COMPILED	Load the compiled version of the file.
	If `COMPILED` is specified, the compiled version will be loaded, if it is found. The source will not be loaded. If neither `SOURCE` or `COMPILED` is specified, the compiled version of the file will be loaded if it is found, otherwise the source will be loaded if it is found.
LOAD	Load the file by calling `LOAD`, if it has not already been loaded. This is the default unless `LOADCOMP` or `LOADFROM` is specified.
	If `LOAD` is specified, `FILESLOAD` considers that the file has already been loaded if the root name of the file has a non-`NIL` `FILEDATES` property. This is a somewhat different algorithm than `LOAD?` uses. In particular, `FILESLOAD` will not load a newer version of a file that has already been loaded.
LOADCOMP	Load the file with `LOADCOMP?` rather than `LOAD`. Automatically implies `SOURCE`.
LOADFROM	Load the file with `LOADFROM` rather than `LOAD`.
NIL, T, PROP **ALLPROP** **SYSLOAD**	The loading function is called with its *LDFLG* argument set to the specified token. *LDFLG* affects the operation of the loading functions by resetting `DFNFLG` (see Chapter 10) to *LDFLG* during the loading. If none of these tokens are specified, the value of the variable `LDFLG` is used if it is bound, otherwise `NIL` is used.
NOERROR	If `NOERROR` is specified, no error occurs when a file is not found.

Each list determines how all further files in the lists are loaded, unless changed by another list. The tokens above can be joined together in a single list. For example,

```
(FILESLOAD (LOADCOMP) NET (SYSLOAD FROM VALUEOF
NEWDIRECTORIES) CJSYS)
```

will call `LOADCOMP?` to load the file `NET` searching the value of `DIRECTORIES`, and then call `LOADCOMP?` to load the file `CJSYS` with *LDFLG* set to `SYSLOAD`, searching the directory list that is the value of the variable `NEWDIRECTORIES`.

FILESLOAD expressions can be dumped on files using the FILES file manager command.

Storing Files

(**MAKEFILE** *FILE OPTIONS REPRINTFNS SOURCEFILE*)　　　　　　　　　　　[Function]

>Makes a new version of the file *FILE*, storing the information specified by *FILE*'s filecoms.　Notices *FILE* if not previously noticed.　Then, it adds *FILE* to NOTLISTEDFILES and NOTCOMPILEDFILES.

>*OPTIONS* is a symbol or list of symbols which specify options.　By specifying certain options, MAKEFILE can automatically compile or list *FILE*.　Note that if *FILE* does not contain any function definitions, it is not compiled even when *OPTIONS* specifies C or RC. The options are spelling corrected using the list MAKEFILEOPTIONS.　If spelling correction fails, MAKEFILE generates an error.　The options are interpreted as follows:

>>**C**
>>**RC**　After making *FILE*, MAKEFILE will compile *FILE* by calling TCOMPL (if C is specified) or RECOMPILE (if RC is specified). If there are any block declarations specified in the filecoms for *FILE*, BCOMPL or BRECOMPILE will be called instead.

>>　　　If F, ST, STF, or S is the *next* item on *OPTIONS* following C or RC, it is given to the compiler as the answer to the compiler's question LISTING?　(see Chapter 18).　For example, (MAKEFILE 'FOO '(C F LIST)) will dump FOO, then TCOMPL or BCOMPL it specifying that functions are not to be redefined, and finally list the file.

>>**LIST**　After making *FILE*, MAKEFILE calls LISTFILES to print a hardcopy listing of *FILE*.

>>**CLISPIFY**　MAKEFILE calls PRETTYDEF with CLISPIFYPRETTYFLG = T (see Chapter 21).　This causes CLISPIFY to be called on each function defined as an EXPR before it is prettyprinted.

>>　　　Alternatively, if *FILE* has the property FILETYPE with value CLISP or a list containing CLISP, PRETTYDEF is called with CLISPIFYPRETTYFLG reset to CHANGES, which will cause CLISPIFY to be called on all functions marked as having been changed.　If *FILE* has property FILETYPE with value CLISP, the compiler will DWIMIFY its functions before compiling them (see Chapter 18).

>>**FAST**　MAKEFILE calls PRETTYDEF with PRETTYFLG = NIL (see Chapter 26).　This causes data objects to be printed rather than prettyprinted, which is much faster.

REMAKE   MAKEFILE "remakes" *FILE*: The prettyprinted definitions of functions that have not changed are copied from an earlier version of the symbolic file. Only those functions that have changed are prettyprinted.

NEW   MAKEFILE does *not* remake *FILE*. If MAKEFILEREMAKEFLG = T (the initial setting), the default for all calls to MAKEFILE is to remake. The NEW option can be used to override this default.

REPRINTFNS and *SOURCEFILE* are used when remaking a file.

FILE is not added to NOTLISTEDFILES if *FILE* has on its property list the property FILETYPE with value DON'TLIST, or a list containing DON'TLIST. *FILE* is not added to NOTCOMPILEDFILES if *FILE* has on its property list the property FILETYPE with value DON'TCOMPILE, or a list containing DON'TCOMPILE. Also, if *FILE* does not contain any function definitions, it is not added to NOTCOMPILEDFILES, and it is not compiled even when *OPTIONS* specifies C or RC.

If a remake is *not* being performed, MAKEFILE checks the state of *FILE* to make sure that the entire source file was actually LOADed. If *FILE* was loaded as a compiled file, MAKEFILE prints the message CAN'T DUMP: ONLY THE COMPILED FILE HAS BEEN LOADED. Similarly, if only some of the symbolic definitions were loaded via LOADFNS or LOADFROM, MAKEFILE prints CAN'T DUMP: ONLY SOME OF ITS SYMBOLICS HAVE BEEN LOADED. In both cases, MAKEFILE will then ask you if it should dump anyway; if you decline, MAKEFILE does not call PRETTYDEF, but simply returns (*FILE* NOT DUMPED) as its value.

You can indicate that *FILE* must be block compiled together with other files as a unit by putting a list of those files on the property list of each file under the property FILEGROUP. If *FILE* has a FILEGROUP property, the compiler will not be called until all files on this property have been dumped that need to be.

MAKEFILE operates by rebinding PRETTYFLG, PRETTYTRANFLG, and CLISPIFYPRETTYFLG, evaluating each expression on MAKEFILEFORMS (under errorset protection), and then calling PRETTYDEF.

PRETTYDEF calls PRETTYPRINT with its second argument *PRETTYDEFLG* = T, so whenever PRETTYPRINT (and hence MAKEFILE) start printing a new function, the name of that function is printed if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

(**MAKEFILES** *OPTIONS FILES*) [Function]

Performs (MAKEFILE *FILE OPTIONS*) for each file on *FILES* that needs to be dumped. If *FILES* = NIL, FILELST is used. For example, (MAKEFILES 'LIST) will make and list all files that have been changed. In this case, if any typed definitions for any items have been defined or changed and they are *not* contained in one of the files on FILELST, MAKEFILES calls ADDTOFILES? to allow you to specify where these go. MAKEFILES returns a list of all files that are made.

(**CLEANUP** *FILE₁ FILE₂ ... FILEₙ*) [NLambda NoSpread Function]

Dumps, lists, and recompiles (with RECOMPILE or BRECOMPILE) any of the specified files (unevaluated) requiring the corresponding operation. If no files are specified, FILELST is used. CLEANUP returns NIL.

CLEANUP uses the value of the variable CLEANUPOPTIONS as the *OPTIONS* argument to MAKEFILE. CLEANUPOPTIONS is initially (RC), to indicate that the files should be recompiled. If CLEANUPOPTIONS is set to (RC F), no listing will be performed, and no functions will be redefined as the result of compiling. Alternatively, if *FILE₁* is a list, it will be interpreted as the list of options regardless of the value of CLEANUPOPTIONS.

(**FILES?**) [Function]

Prints on the terminal the names of those files that have been modified but not dumped, dumped but not listed, dumped but not compiled, plus the names of any functions and other typed definitions (if any) that are not contained in any file. If there are any, FILES? then calls ADDTOFILES? to allow you to specify where these go.

(**ADDTOFILES?** —) [Function]

Called from MAKEFILES, CLEANUP, and FILES? when there are typed definitions that have been marked as changed which do not belong to any file. ADDTOFILES? lists the names of the changed items, and asks if you want to specify where these items should be put. If you answer N(o), ADDTOFILES? returns NIL without taking any action. If you answer], this is taken to be an answer to each question that would be asked, and all the changed items are marked as dummy items to be ignored. Otherwise, ADDTOFILES? prints the name of each changed item, and accepts one of the following responses:

> A file name
> A filevar   If you give a file name or a variable whose value is a list (a filevar), the item is added to the corresponding file or list, using ADDTOFILE.
>
> If your response is not the name of a file on FILELST or a variable whose value is a list, you will be asked whether it is a new file. If you say no, then ADDTOFILES? will check whether the item is the name of a list, i.e., whether its value is a list. If not, youwill be asked whether it is a new list.

line-feed	Same as your previous response.
space carriage return	Take no action.
]	The item is marked as a dummy item by adding it to `NILCOMS`. This tells the file manager simply to ignore this item.
[	The "definition" of the item in question is prettyprinted to the terminal, and then you are asked again about its disposition.
(	`ADDTOFILES?` prompts with "`LISTNAME: (`", you type in the name of a list, i.e. a variable whose value is a list, terminated by a `)`. The item will then only be added to (under) a command in which the named list appears as a filevar. If none are found, a message is printed, and you are asked again. For example, you define a new function `FOO3`. When asked where it goes, you type `(FOOFNS)`. If the command `(FNS * FOOFNS)` is found, `FOO3` will be added to the value of `FOOFNS`. If instead you type `(FOOCOMS)`, and the command `(COMS * FOOCOMS)` is found, then `FOO3` will be added to a command for dumping functions that is contained in `FOOCOMS`. If the named list is not also the name of a file, you can simply type it in without parenthesis as described above.
@	`ADDTOFILES?` prompts with "`Near: (`", you type in the name of an object, and the item is then inserted in a command for dumping objects (of its type) that contains the indicated name. The item is inserted immediately after the indicated name.

(**LISTFILES** *FILE₁ FILE₂ ... FILEₙ*)   [NLambda NoSpread Function]

Lists each of the specified files (unevaluated). If no files are given, `NOTLISTEDFILES` is used. Each file listed is removed from `NOTLISTEDFILES` if the listing is completed. For each file not found, `LISTFILES` prints the message *FILENAME* `NOT FOUND` and proceeds to the next file.

`LISTFILES` calls the function `LISTFILES1` on each file to be listed. Normally, `LISTFILES1` is defined to simply call `SEND.FILE.TO.PRINTER` (see Chapter 29), but you can advise or redefine `LISTFILES1` for more specialized applications.

Any lists inside the argument list to `LISTFILES` are interpreted as property lists that set the various printing options, such as the printer, number of copies, banner page name, etc (see see Chapter 29). Later properties override earlier ones. For example,

        (LISTFILES FOO (HOST JEDI) FUM (#COPIES 3) FIE)

will cause one copy of FOO to be printed on the default printer, and one copy of FUM and three copies of FIE to be printed on the printer JEDI.

(**COMPILEFILES** $FILE_1$ $FILE_2$ ... $FILE_N$)   [NLambda NoSpread Function]

Executes the RC and C options of MAKEFILE for each of the specified files (unevaluated). If no files are given, NOTCOMPILEDFILES is used. Each file compiled is removed from NOTCOMPILEDFILES. If $FILE_1$ is a list, it is interpreted as the OPTIONS argument to MAKEFILES. This feature can be used to supply an answer to the compiler's LISTING? question, e.g., (COMPILEFILES (STF)) will compile each file on NOTCOMPILEDFILES so that the functions are redefined without the EXPRs definitions being saved.

(**WHEREIS** NAME TYPE FILES FN)   [Function]

TYPE is a file manager type. WHEREIS sweeps through all the files on the list FILES and returns a list of all files containing NAME as a TYPE. WHEREIS knows about and expands all file manager commands and file manager macros. TYPE = NIL defaults to FNS (to retrieve function definitions). If FILES is not a list, the value of FILELST is used.

If FN is given, it should be a function (with arguments NAME, FILE, and TYPE) which is applied for every file in FILES that contains NAME as a TYPE. In this case, WHEREIS returns NIL.

If the WHEREIS library package has been loaded, WHEREIS is redefined so that FILES = T means to use the whereis package data base, so WHEREIS will find NAME even if the file has not been loaded or noticed. FILES = NIL always means use FILELST.

Remaking a Symbolic File

Most of the time that a symbolic file is written using MAKEFILE, only a few of the functions that it contains have been changed since the last time the file was written. Rather than prettprinting all of the functions, it is often considerably faster to "remake" the file, copying the prettprinted definitions of unchanged functions from an earlier version of the symbolic file, and only prettyprinting those functions that have been changed.

MAKEFILE will remake the symbolic file if the REMAKE option is specified. If the NEW option is given, the file is not remade, and all of the functions are prettprinted. The default action is specified by the value of MAKEFILEREMAKEFLG: if T (its initial value), MAKEFILE will remake files unless the NEW option is given; if NIL, MAKEFILE will not remake unless the REMAKE option is given.

Note: If the file has never been loaded or dumped, for example if the filecoms were simply set up in memory, then MAKEFILE will never attempt to remake the file, regardless of the setting of MAKEFILEREMAKEFLG, or whether the REMAKE option was specified.

When MAKEFILE is remaking a symbolic file, you can explicitly indicate the functions which are to be prettyprinted and the file to be used for copying the rest of the function definitions from via the *REPRINTFNS* and *SOURCEFILE* arguments to MAKEFILE. Normally, both of these arguments are defaulted to NIL. In this case, *REPRINTFNS* will be set to those functions that have been changed since the last version of the file was written. For *SOURCEFILE*, MAKEFILE obtains the full name of the most recent version of the file (that it knows about) from the FILEDATES property of the file, and checks to make sure that the file still exists and has the same file date as that stored on the FILEDATES property. If it does, MAKEFILE uses that file as *SOURCEFILE*. This procedure permits you to LOAD or LOADFROM a file in a different directory, and still be able to remake the file with MAKEFILE. In the case where the most recent version of the file cannot be found, MAKEFILE will attempt to remake using the *original* version of the file (i.e., the one first loaded), specifying as *REPRINTFNS* the union of all changes that have been made since the file was first loaded, which is obtained from the FILECHANGES property of the file. If both of these fail, MAKEFILE prints the message "CAN'T FIND EITHER THE PREVIOUS VERSION OR THE ORIGINAL VERSION OF *FILE*, SO IT WILL HAVE TO BE WRITTEN ANEW", and does not remake the file, i.e. will prettyprint all of the functions.

When a remake is specified, MAKEFILE also checks to see how the file was originally loaded. If the file was originally loaded as a compiled file, MAKEFILE will call LOADVARS to obtain those DECLARE: expressions that are contained on the symbolic file, but not the compiled file, and hence have not been loaded. If the file was loaded by LOADFNS (but not LOADFROM), then LOADVARS is called to obtain any non-DEFINEQ expressions. Before calling LOADVARS to re-load definitions, MAKEFILE asks you, e.g. "Only the compiled version of FOO was loaded, do you want to LOADVARS the (DECLARE: .. DONTCOPY ..) expressions from {DSK}<MYDIR>FOO.;3?". You can respond Yes to execute the LOADVARS and continue the MAKEFILE, No to proceed with the MAKEFILE without performing the LOADVARS, or Abort to abort the MAKEFILE. You may wish to skip the LOADVARS if you had circumvented the file manager in some way, and loading the old definitions would overwrite new ones.

Remaking a symbolic file is considerably faster if the earlier version has a *file map* indicating where the function definitions are located (see the File Maps section), but it does not depend on this information.

Loading Files in a Distributed Environment

Each Interlisp source and compiled code file contains the full filename of the file, including the host and directory names, in a FILECREATED expression at the beginning of the file. The compiled code file also contains the full file name of the source file it was created from. In earlier versions of Interlisp, the file manager used this information to locate the appropriate source file when "remaking" or recompiling a file.

This turned out to be a bad feature in distributed environments, where users frequently move files from one place to another, or where files are stored on removable media. For example, suppose you MAKEFILE to a floppy, and then copy the file to a file server. If you loaded and edited the file from a file server, and tried to do MAKEFILE, it would try to locate the source file on the floppy, which is probably no longer loaded.

Currently, the file manager searches for sources file on the connected directory, and on the directory search path (on the variable DIRECTORIES). If it is not found, the host/directory information from the FILECREATED expression be used.

> Warning: One situation where the new algorithm does the wrong thing is if you explicitly LOADFROM a file that is not on your directory search path. Future MAKEFILEs and CLEANUPs will search the connected directory and DIRECTORIES to find the source file, rather than using the file that the LOADFROM was done from. Even if the correct file is on the directory search path, you could still create a bad file if there is another version of the file in an earlier directory on the search path. In general, you should either explicitly specify the *SOURCEFILE* argument to MAKEFILE to tell it where to get the old source, or connect to the directory where the correct source file is.

Marking Changes

The file manager needs to know what typed definitions have been changed, so it can determine which files need to be updated. This is done by "marking changes". All the system functions that perform file manager operations (LOAD, TCOMPL, PRETTYDEF, etc.), as well as those functions that define or change data, (EDITF, EDITV, EDITP, DWIM corrections to user functions) interact with the file manager by marking changes. Also, *typed-in* assignment of variables or property values is noticed by the file manager. (If a program modifies a variable or property value, this is not noticed.) In some cases the marking procedure can be subtle, e.g. if you edit a property list using EDITP, only those properties whose values are actually changed (or added) are marked.

The various system functions which create or modify objects call MARKASCHANGED to mark the object as changed. For example, when a function is defined via DEFINE or DEFINEQ, or modified via EDITF, or a DWIM correction, the function is marked as being a changed object of type FNS. Similarly, whenever a new record is declared, or an existing record redeclared or edited, it is marked as being a changed object of type RECORDS, and so on for all of the other file manager types.

You can also call MARKASCHANGED directly to mark objects of a particular file manager type as changed:

(**MARKASCHANGED** *NAME TYPE REASON*)   [Function]

> Marks *NAME* of type *TYPE* as being changed. MARKASCHANGED returns *NAME*. MARKASCHANGED is undoable.

> *REASON* is a symbol that indicated how *NAME* was changed. MARKASCHANGED recognizes the following values for *REASON*:

>> **DEFINED**   Used to indicate the creation of *NAME*, e.g. from DEFINEQ (Chapter 10).

>> **CHANGED**   Used to indicate a change to *NAME*, e.g. from the editor.

DELETED	Used to indicate the deletion of *NAME*, e.g. by DELDEF.
CLISP	Used to indicate the modification of *NAME* by CLISP translation.

For backwards compatibility, MARKASCHANGED also accepts a *REASON* of T (=DEFINED) and NIL (=CHANGED). New programs should avoid using these values.

The variable MARKASCHANGEDFNS is a list of functions that MARKASCHANGED calls (with arguments *NAME*, *TYPE*, and *REASON*). Functions can be added to this list to "advise" MARKASCHANGED to do additional work for all types of objects. The WHENCHANGED file manager type property (see the Defining New File Manager Types section) can be used to specify additional actions when MARKASCHANGED gets called on specific types of objects.

(**UNMARKASCHANGED** *NAME TYPE*) [Function]

Unmarks *NAME* of type *TYPE* as being changed. Returns *NAME* if *NAME* was marked as changed and is now unmarked, NIL otherwise. UNMARKASCHANGED is undoable.

(**FILEPKGCHANGES** *TYPE LST*) [NoSpread Function]

If *LST* is not specified (as opposed to being NIL), returns a list of those objects of type *TYPE* that have been marked as changed but not yet associated with their corresponding files (see the File Manager Types section). If *LST* is specified, FILEPKGCHANGES sets the corresponding list. (FILEPKGCHANGES) returns a list of *all* objects marked as changed as a list of elements of the form (*TYPENAME . CHANGEDOBJECTS*).

Some properties (e.g. EXPR, ADVICE, MACRO, I.S.OPR, etc.) are used to implement other file manager types. For example, if you change the value of the property I.S.OPR, you are really changing an object of type I.S.OPR. The effect is the same as though you had redefined the i.s.opr via a direct call to the function I.S.OPR. If a property whose value has been changed or added does not correspond to a specific file manager type, then it is marked as a changed object of type PROPS whose *name* is (*VARIABLENAME PROPNAME*) (except if the property name has a property PROPTYPE with value IGNORE).

Similarly, if you change a variable which implements the file manager type ALISTS (as indicated by the appearance of the property VARTYPE with value ALIST on the variable's property list), only those entries that are actually changed are marked as being changed objects of type ALISTS. The "name" of the object will be (*VARIABLENAME KEY*) where *KEY* is CAR of the entry on the alist that is being marked. If the variable corresponds to a specific file manager type other than ALISTS, e.g., USERMACROS, LISPXMACROS, etc., then an object of that type is marked. In this case, the name of the changed object will be CAR of the corresponding entry on the alist. For example, if you edit LISPXMACROS and change a definition for PL, then the object PL of type LISPXMACROS is marked as being changed.

Noticing Files

Already existing files are "noticed" by `LOAD` or `LOADFROM` (or by `LOADFNS` or `LOADVARS` when the *VARS* argument is `T`. New files are noticed when they are constructed by `MAKEFILE`, or when definitions are first associated with them via `FILES?` or `ADDTOFILES?`. Noticing a file updates certain lists and properties so that the file manager functions know to include the file in their operations. For example, `CLEANUP` will only dump files that have been noticed.

You can explicitly tell the file manager to notice a newly-created file by defining the filecoms for the file, and calling `ADDFILE`:

(**ADDFILE** *FILE*)   [Function]

> Tells the file manager that *FILE* should be recognized as a file; it adds *FILE* to `FILELST`, and also sets up the `FILE` property of *FILE* to reflect the current set of changes which are "registered against" *FILE*.

The file manager uses information stored on the property list of the root name of noticed files. The following property names are used:

FILE   [Property Name]

> When a file is noticed, the property `FILE`, value ((*FILECOMS* . *LOADTYPE*)) is added to the property list of its root name. *FILECOMS* is the variable containing the filecoms of the file. *LOADTYPE* indicates *how* the file was loaded, e.g., completely loaded, only partially loaded as with `LOADFNS`, loaded as a compiled file, etc.

> The property `FILE` is used to determine whether or not the corresponding file has been modified since the last time it was loaded or dumped. `CDR` of the `FILE` property records by type those items that have been changed since the last `MAKEFILE`. Whenever a file is dumped, these items are moved to the property `FILECHANGES`, and `CDR` of the `FILE` property is reset to `NIL`.

FILECHANGES   [Property Name]

> The property `FILECHANGES` contains a list of all changed items since the file was loaded (there may have been several sequences of editing and rewriting the file). When a file is dumped, the changes in `CDR` of the `FILE` property are added to the `FILECHANGES` property.

FILEDATES   [Property Name]

> The property `FILEDATES` contains a list of version numbers and corresponding file dates for this file. These version numbers and dates are used for various integrity checks in connection with *re*making a file.

FILEMAP [Property Name]

The property `FILEMAP` is used to store the filemap for the file. This is used to directly load individual functions from the middle of a file.

To compute the root name, `ROOTFILENAME` is applied to the name of the file as indicated in the `FILECREATED` expression appearing at the front of the file, since this name corresponds to the name the file was originally made under. The file manager detects that the file being noticed is a compiled file (regardless of its name), by the appearance of more than one `FILECREATED` expressions. In this case, each of the files mentioned in the following `FILECREATED` expressions are noticed. For example, if you perform `(BCOMPL '(FOO FIE))`, and subsequently loads `FOO.DCOM`, both `FOO` and `FIE` will be noticed.

When a file is noticed, its root name is added to the list `FILELST`:

FILELST [Variable]

Contains a list of the root names of the files that have been noticed.

LOADEDFILELST [Variable]

Contains a list of the actual names of the files as loaded by `LOAD`, `LOADFNS`, etc. For example, if you perform `(LOAD '<NEWLISP>EDITA.COM;3)`, `EDITA` will be added to `FILELST`, but `<NEWLISP>EDITA.COM;3` is added to `LOADEDFILELST`. `LOADEDFILELST` is not used by the file manager; it is maintained solely for your benefit.

Distributing Change Information

Periodically, the function `UPDATEFILES` is called to find which file(s) contain the elements that have been changed. `UPDATEFILES` is called by `FILES?`, `CLEANUP`, and `MAKEFILES`, i.e., any procedure that requires the `FILE` property to be up to date. This procedure is followed rather than updating the `FILE` property after each change because scanning `FILELST` and examining each file manager command can be a time-consuming process; this is not so noticeable when performed in conjunction with a large operation like loading or writing a file.

`UPDATEFILES` operates by scanning `FILELST` and interrogating the file manager commands for each file. When (if) any files are found that contain the corresponding typed definition, the name of the element is added to the value of the property `FILE` for the corresponding file. Thus, after `UPDATEFILES` has completed operating, the files that need to be dumped are simply those files on `FILELST` for which `CDR` of their `FILE` property is non-`NIL`. For example, if you load the file `FOO` containing definitions for `FOO1`, `FOO2`, and `FOO3`, edit `FOO2`, and then call `UPDATEFILES`, `(GETPROP 'FOO 'FILE)` will be `((FOOCOMS . T) (FNS FOO2))`. If any objects marked as changed have not been transferred to the `FILE` property for some file, e.g., you define a new function but forget (or declines) to add it to the file manager commands for the corresponding file, then both `FILES?` and

CLEANUP will print warning messages, and then call ADDTOFILES? to permit you to specify on which files these items belong.

You can also invoke UPDATEFILES directly:

(**UPDATEFILES** — —) [Function]

(UPDATEFILES) will update the FILE properties of the noticed files.

File Manager Types

In addition to the definitions of functions and values of variables, source files in Interlisp can contain a variety of other information, e.g. property lists, record declarations, macro definitions, hash arrays, etc. In order to treat such a diverse assortment of data uniformly from the standpoint of file operations, the file manager uses the concept of a *typed definition*, of which a function definition is just one example. A typed definition associates with a name (usually a symbol), a definition of a given type (called the file manager type). Note that the same name may have several definitions of different types. For example, a symbol may have both a function definition and a variable definition. The file manager also keeps track of the file that a particular typed definition is stored on, so one can think of a typed definition as a relation between four elements: a name, a definition, a type, and a file.

A file manager type is an abstract notion of a class of objects which share the property that every object of the same file manager type is stored, retrieved, edited, copied etc., by the file manager in the same way. Each file manager type is identified by a symbol, which can be given as an argument to the functions that manipulate typed definitions. You may define new file manager types, as described in the Defining New Package Types section.

FILEPKGTYPES [Variable]

The value of FILEPKGTYPES is a list of all file manager types, including any that you may have defined.

The file manager is initialized with the following built-in file manager types:

ADVICE [File Manager Type]

Used to access "advice" modifying a function (see Chapter 15).

ALISTS [File Manager Type]

Used to access objects stored on an association list that is the value of a symbol (see Chapter 3).

A variable is declared to have an association list as its value by putting on its property list the property VARTYPE with value ALIST. In this case, each dotted pair on the list is an object of type ALISTS. When the value of such a variable is changed, only those entries in the association list that are actually changed or added are marked as changed objects of

type `ALISTS` (with "name" (`SYMBOL` *KEY*)). Objects of type `ALISTS` are dumped via the `ALISTS` or `ADDVARS` file manager commands.

Note that some association lists are used to "implement" other file manager types. For example, the value of the global variable `USERMACROS` implements the file manager type `USERMACROS` and the values of `LISPXMACROS` and `LISPXHISTORYMACROS` implement the file manager type `LISPXMACROS`. This is indicated by putting on the property list of the variable the property `VARTYPE` with value a list of the form (`ALIST` *FILEPKGTYPE*). For example, (`GETPROP 'LISPXHISTORYMACROS 'VARTYPE`) => (`ALIST LISPXMACROS`).

COURIERPROGRAMS   [File Manager Type]

Used to access Courier programs (see Chapter 31).

EXPRESSIONS   [File Manager Type]

Used to access lisp expressions that are put on a file by using the `REMEMBER` programmers assistant command (Chapter 13), or by explicitly putting the `P` file manager command on the filecoms.

FIELDS   [File Manager Type]

Used to access fields of records. The "definition" of an object of type `FIELDS` is a list of all the record declarations which contain the name. See Chapter 8.

FILEPKGCOMS   [File Manager Type]

Used to access file manager commands and types. A single name can be defined both as a file manager type and a file manager command. The "definition" of an object of type `FILEPKGCOMS` is a list structure of the form ((`COM` . *COMPROPS*) (`TYPE` . *TYPEPROPS*)), where *COMPROPS* is a property list specifying how the name is defined as a file manager command by `FILEPKGCOM` (see the Defining New File Manager Commands section), and *TYPEPROPS* is a property list specifying how the name is defined as a file manager type by `FILEPKGTYPE` (see the Defining New File Manager Types section).

FILES   [File Manager Type]

Used to access files. This file manager type is most useful for renaming files. The "definition" of a file is not a useful structure.

FILEVARS   [File Manager Type]

Used to access Filevars (see the FileVars section).

FNS   [File Manager Type]

Used to access function definitions.

I.S.OPRS [File Manager Type]

Used to access the definitions of iterative statement operators (see Chapter 9).

LISPXMACROS [File Manager Type]

Used to access programmer's assistant commands defined on the variables `LISPXMACROS` and `LISPXHISTORYMACROS` (see Chapter 13).

MACROS [File Manager Type]

Used to access macro definitions (see Chapter 10).

PROPS [File Manager Type]

Used to access objects stored on the property list of a symbol (see Chapter 2). When a property is changed or added, an object of type PROPS, with "name" (SYMBOL *PROPNAME*) is marked as being changed.

Note that some symbol properties are used to implement other file manager types. For example, the property MACRO implements the file manager type MACROS, the property ADVICE implements ADVICE, etc. This is indicated by putting the property PROPTYPE, with value of the file manager type on the property list of the property name. For example, (GETPROP 'MACRO 'PROPTYPE) => MACROS. When such a property is changed or added, an object of the corresponding file manager type is marked. If (GETPROP *PROPNAME* 'PROPTYPE) => IGNORE, the change is ignored. The FILE, FILEMAP, FILEDATES, etc. properties are all handled this way. (IGNORE cannot be the name of a file manager type implemented as a property).

RECORDS [File Manager Type]

Used to access record declarations (see Chapter 8).

RESOURCES [File Manager Type]

Used to access resources (see Chapter 12).

TEMPLATES [File Manager Type]

Used to access Masterscope templates (see Chapter 19).

USERMACROS [File Manager Type]

Used to access user edit macros (see Chapter 16).

VARS [File Manager Type]

Used to access top-level variable values.

Functions for Manipulating Typed Definitions

The functions described below can be used to manipulate typed definitions, without needing to know how the manipulations are done. For example, (GETDEF 'FOO 'FNS) will return the function definition of FOO, (GETDEF 'FOO 'VARS) will return the variable value of FOO, etc. All of the functions use the following conventions:

1. All functions which make destructive changes are undoable.

2. Any argument that expects a list of symbols will also accept a single symbol, operating as though it were enclosed in a list. For example, if the argument *FILES* should be a list of files, it may also be a single file.

3. *TYPE* is a file manager type. *TYPE* = NIL is equivalent to *TYPE* = FNS. The singular form of a file manager type is also recognized, e.g. *TYPE* = VAR is equivalent to *TYPE* = VARS.

4. *FILES* = NIL is equivalent to *FILES* = FILELST.

5. *SOURCE* is used to indicate the source of a definition, that is, where the definition should be found. *SOURCE* can be one of:

   CURRENT   Get the definition currently in effect.

   SAVED   Get the "saved" definition, as stored by SAVEDEF.

   FILE   Get the definition contained on the (first) file determined by WHEREIS.

      WHEREIS is called with *FILES* = T, so that if the WHEREIS library package is loaded, the WHEREIS data base will be used to find the file containing the definition.

   ?   Get the definition currently in effect if there is one, else the saved definition if there is one, otherwise the definition from a file determined by WHEREIS. Like specifying CURRENT, SAVED, and FILE in order, and taking the first definition that is found.

   a file name
   a list of file names   Get the definition from the first of the indicated files that contains one.

   NIL   In most cases, giving *SOURCE* = NIL (or not specifying it at all) is the same as giving ?, to get either the current, saved, or filed definition. However, with HASDEF, *SOURCE* = NIL is interpreted as equal to *SOURCE* = CURRENT, which only tests if there is a current definition.

The operation of most of the functions described below can be changed or extended by modifying the appropriate properties for the corresponding file manager type using the function FILEPKGTYPE, described in the Defining New File Manager Types section.

(**GETDEF** *NAME TYPE SOURCE OPTIONS*)   [Function]

Returns the definition of *NAME*, of type *TYPE*, from *SOURCE*. For most types, GETDEF returns the expression which would be pretty printed when dumping *NAME* as *TYPE*. For example, for *TYPE* = FNS, an EXPR definition is returned, for *TYPE* = VARS, the value of *NAME* is returned, etc.

OPTIONS is a list which specifies certain options:

NOERROR   GETDEF causes an error if an appropriate definition cannot be found, unless *OPTIONS* is or contains NOERROR. In this case, GETDEF returns the value of the NULLDEF file manager type property (see the Defining New File Manager Types section), usually NIL.

a string   If *OPTIONS* is or contains a string, that string will be returned if no definition is found (and NOERROR is not among the options). The caller can thus determine whether a definition was found, even for types for which NIL or NOBIND are acceptable definitions.

NOCOPY   GETDEF returns a copy of the definition unless *OPTIONS* is or contains NOCOPY.

EDIT   If *OPTIONS* is or contains EDIT, GETDEF returns a copy of the definition unless it is possible to edit the definition "in place." With some file manager types, such as functions, it is meaningful (and efficient) to edit the definition by destructively modifying the list structure, without calling PUTDEF. However, some file manager types (like records) need to be "installed" with PUTDEF after they are edited. The default EDITDEF (see the Defining New File Manager Types section) calls GETDEF with *OPTIONS* of (EDIT NOCOPY), so it doesn't use a copy unless it has to, and only calls PUTDEF if the result of editing is not EQUAL to the old definition.

NODWIM   A FNS definition will be dwimified if it is likely to contain CLISP unless *OPTIONS* is or contains NODWIM.

(**PUTDEF** *NAME TYPE DEFINITION REASON*)   [Function]

Defines *NAME* of type *TYPE* with *DEFINITION*. For *TYPE* = FNS, does a DEFINE; for *TYPE* = VARS, does a SAVESET, etc.

For *TYPE* = FILES, PUTDEF establishes the command list, notices *NAME*, and then calls MAKEFILE to actually dump the file *NAME*, copying functions if necessary from the "old" file (supplied as part of *DEFINITION*).

PUTDEF calls MARKASCHANGED (see the Mrking Changes section) to mark *NAME* as changed, giving a reason of *REASON*. If *REASON* is NIL, the default is DEFINED.

If *TYPE* = FNS, PUTDEF prints a warning if you try to redefine a function on the list UNSAFE.TO.MODIFY.FNS (see Chapter 10).

(**HASDEF** *NAME TYPE SOURCE SPELLFLG*)   [Function]

Returns (OR *NAME* T) if *NAME* is the name of something of type *TYPE*. If not, attempts spelling correction if *SPELLFLG* = T, and returns the spelling-corrected *NAME*. Otherwise returns NIL. HASDEF for type FNS (or NIL) indicates that *NAME* has an editable source definition. If *NAME* is a function that exists on a file for which you have loaded only the compiled version and not the source, HASDEF returns NIL.

(HASDEF NIL *TYPE*) returns T if NIL has a valid definition.

If *SOURCE* = NIL, *HASDEF* interprets this as equal to *SOURCE* = CURRENT, which only tests if there is a current definition.

(**TYPESOF** *NAME POSSIBLETYPES IMPOSSIBLETYPES SOURCE*)   [Function]

Returns a list of the types in *POSSIBLETYPES* but not in *IMPOSSIBLETYPES* for which *NAME* has a definition. FILEPKGTYPES is used if *POSSIBLETYPES* is NIL.

(**COPYDEF** *OLD NEW TYPE SOURCE OPTIONS*)   [Function]

Defines *NEW* to have a copy of the definition of *OLD* by doing PUTDEF on a copy of the definition retrieved by (GETDEF *OLD TYPE SOURCE OPTIONS*). *NEW* is substituted for *OLD* in the copied definition, in a manner that may depend on the *TYPE*.

For example, (COPYDEF 'PDQ 'RST 'FILES) sets up RSTCOMS to be a copy of PDQCOMS, changes things like (VARS * PDQVARS) to be (VARS * RSTVARS) in RSTCOMS, and performs a MAKEFILE on RST such that the appropriate definitions get copied from PDQ.

COPYDEF disables the NOCOPY option of GETDEF, so *NEW* will always have a *copy* of the definition of *OLD*.

COPYDEF substitutes *NEW* for *OLD* throughout the definition of *OLD*. This is usually the right thing to do, but in some cases, e.g., where the old name appears within a quoted expression but was not used in the same context, you must re-edit the definition.

(**DELDEF** *NAME TYPE*)   [Function]

Removes the definition of *NAME* as a *TYPE* that is currently in effect.

(**SHOWDEF** *NAME TYPE FILE*)   [Function]

> Prettyprints the definition of *NAME* as a *TYPE* to *FILE*. This shows you how *NAME* would be written to a file. Used by ADDTOFILES? (see the Storing Files section).

(**EDITDEF** *NAME TYPE SOURCE EDITCOMS*)   [Function]

> Edits the definition of *NAME* as a *TYPE*. Essentially performs

```
(PUTDEF NAME TYPE
    (EDITE (GETDEF NAME TYPE SOURCE)
           EDITCOMS))
```

(**SAVEDEF** *NAME TYPE DEFINITION*)   [Function]

> Sets the "saved" definition of *NAME* as a *TYPE* to *DEFINITION*. If *DEFINITION* = NIL, the current definition of *NAME* is saved.

> If *TYPE* = FNS (or NIL), the function definition is saved on *NAME*'s property list under the property EXPR, or CODE (depending on the FNTYP of the function definition). If (GETD *NAME*) is non-NIL, but (FNTYP *FN*) = NIL, SAVEDEF saves the definition on the property name LIST. This can happen if a function was somehow defined with an illegal expr definition, such as (LAMMMMDA (X) ...).

> If *TYPE* = VARS, the definition is stored as the value of the VALUE property of *NAME*. For other types, the definition is stored in an internal data structure, from where it can be retrieved by GETDEF or UNSAVEDEF.

(**UNSAVEDEF** *NAME TYPE*)   [Function]

> Restores the "saved" definition of *NAME* as a *TYPE*, making it be the current definition. Returns *PROP*.

> If *TYPE* = FNS (or NIL), UNSAVEDEF unsaves the function definition from the EXPR property if any, else CODE, and returns the property name used. UNSAVEDEF also recognizes *TYPE* = EXPR, CODE, or LIST, meaning to unsave the definition only from the corresponding property only.

> If DFNFLG is not T (see Chapter 10), the current definition of *NAME*, if any, is saved using SAVEDEF. Thus one can use UNSAVEDEF to switch back and forth between two definitions.

(**LOADDEF** *NAME TYPE SOURCE*)   [Function]

> Equivalent to (PUTDEF *NAME TYPE* (GETDEF *NAME TYPE SOURCE*)). LOADDEF is essentially a generalization of LOADFNS, e.g. it enables loading a single record declaration from a file. (LOADDEF *FN*) will give *FN* an EXPR definition, either obtained from its property list or a file, unless it already has one.

(**CHANGECALLERS** *OLD NEW TYPES FILES METHOD*)   [Function]

Finds all of the places where *OLD* is used as any of the types in *TYPES* and changes those places to use *NEW*. For example, (CHANGECALLERS 'NLSETQ 'ERSETQ) will change all calls to NLSETQ to be calls to ERSETQ. Also changes occurrences of *OLD* to *NEW* inside the filecoms of any file, inside record declarations, properties, etc.

CHANGECALLERS attempts to determine if *OLD* might be used as more than one type; for example, if it is both a function and a record field. If so, rather than performing the transformation *OLD -> NEW* automatically, you are allowed to edit all of the places where *OLD* occurs. For each occurrence of *OLD*, you are asked whether you want to make the replacement. If you respond with anything except Yes or No, the editor is invoked on the expression containing that occurrence.

There are two different methods for determining which functions are to be examined. If *METHOD* = EDITCALLERS, EDITCALLERS is used to search *FILES* (see Chapter 16). If *METHOD* = MASTERSCOPE, then the Masterscope database is used instead. *METHOD* = NIL defaults to MASTERSCOPE if the value of the variable DEFAULTRENAMEMETHOD is MASTERSCOPE and a Masterscope database exists, otherwise it defaults to EDITCALLERS.

(**RENAME** *OLD NEW TYPES FILES METHOD*)   [Function]

First performs (COPYDEF *OLD NEW TYPE*) for all *TYPE* inside *TYPES*. It then calls CHANGECALLERS to change all occurrences of *OLD* to *NEW*, and then "deletes" *OLD* with DELDEF. For example, if you have a function FOO which you now wish to call FIE, simply perform (RENAME 'FOO 'FIE), and FIE will be given FOO's definition, and all places that FOO are called will be changed to call FIE instead.

METHOD is interpreted the same as the *METHOD* argument to CHANGECALLERS, above.

(**COMPARE** *NAME$_1$ NAME$_2$ TYPE SOURCE$_1$ SOURCE$_2$*)   [Function]

Compares the definition of *NAME$_1$* with that of *NAME$_2$*, by calling COMPARELISTS (Chapter 3) on (GETDEF *NAME$_1$ TYPE SOURCE$_1$*) and (GETDEF *NAME$_2$ TYPE SOURCE$_2$*), which prints their differences on the terminal.

For example, if the current value of the variable A is (A B C (D E F) G), and the value of the variable B on the file <lisp>FOO is (A B C (D F E) G), then:

```
←(COMPARE 'A 'B 'VARS 'CURRENT '<lisp>FOO)
A from CURRENT and B from <lisp>TEST differ:
(E -> F) (F -> E)
T
```

(**COMPAREDEFS** *NAME TYPE SOURCES*)   [Function]

Calls COMPARELISTS (Chapter 3) on all pairs of definitions of *NAME* as a *TYPE* obtained from the various *SOURCES* (interpreted as a list of source specifications).

Defining New File Manager Types

All manipulation of typed definitions in the file manager is done using the type-independent functions GETDEF, PUTDEF, etc. Therefore, to define a new file manager type, it is only necessary to specify (via the function FILEPKGTYPE) what these functions should do when dealing with a typed definition of the new type. Each file manager type has the following properties, whose values are functions or lists of functions:

These functions are defined to take a *TYPE* argument so that you may have the same function for more than one type.

GETDEF   [File Manager Type Property]

> Value is a function of three arguments, *NAME*, *TYPE*, and *OPTIONS*, which should return the current definition of *NAME* as a type *TYPE*. Used by GETDEF (see the Functions for Manipulating Typed Definitions section), which passes its *OPTIONS* argument.

> If there is no GETDEF property, a file manager command for dumping *NAME* is created (by MAKENEWCOM). This command is then used to write the definition of *NAME* as a type *TYPE* onto the file FILEPKG.SCRATCH (in Medley, this file is created on the {CORE} device). This expression is then read back in and returned as the current definition.

> In some situations, the function HASDEF needs to call GETDEF to determine whether a definition exists. In this case, *OPTIONS* will include the symbol HASDEF, and it is permissable for a GETDEF function to return T or NIL, rather than creating a complex structure which will not be used.

NULLDEF   [File Manager Type Property]

> The value of the NULLDEF property is returned by GETDEF (see the Functions for Manipulating Typed Definitions section) when there is no definition and the NOERROR option is supplied. For example, the NULLDEF of VARS is NOBIND.

FILEGETDEF   [File Manager Type Property]

> This enables you to provide a way of obtaining definitions from a file that is more efficient than the default procedure used by GETDEF (see the Functions for Manipulating Typed Definitions section). Value is a function of four arguments, *NAME*, *TYPE*, *FILE*, and *OPTIONS*. The function is applied by GETDEF when it is determined that a typed definition is needed from a particular file. The function must open and search the given file and return any *TYPE* definition for *NAME* that it finds.

CANFILEDEF   [File Manager Type Property]

> If the value of this property is non-NIL, this indicates that definitions of this file manager type are not loaded when a file is loaded with LOADFROM (see the Loading Files section). The default is NIL. Initially, only FNS has this property set to non-NIL.

PUTDEF [File Manager Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *DEFINITION*, which should store *DEFINITION* as the definition of *NAME* as a type *TYPE*. Used by PUTDEF (see the Functions for Manipulating Typed Definitions section).

HASDEF [File Manager Type Property]

Value is a function of three arguments, *NAME*, *TYPE*, and *SOURCE*, which should return (OR *NAME* T) if *NAME* is the name of something of type *TYPE*. *SOURCE* is as interpreted by HASDEF (see the Functions for Manipulating Typed Definitions section), which uses this property.

EDITDEF [File Manager Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *SOURCE*, and *EDITCOMS*, which should edit the definition of *NAME* as a type *TYPE* from the source *SOURCE*, interpreting the edit commands *EDITCOMS*. If sucessful, should return *NAME* (or a spelling-corrected *NAME*). If it returns NIL, the "default" editor is called. Used by EDITDEF (see the Functions for Manipulating Typed Definitions section).

DELDEF [File Manager Type Property]

Value is a function of two arguments, *NAME*, and *TYPE*, which removes the definition of *NAME* as a *TYPE* that is currently in effect. Used by DELDEF (see the Functions for Manipulating Typed Definitions section).

NEWCOM [File Manager Type Property]

Value is a function of four arguments, *NAME*, *TYPE*, *LISTNAME*, and *FILE*. Specifies how to make a new (instance of a) file manager command to dump *NAME*, an object of type *TYPE*. The function should return the new file manager command. Used by ADDTOFILE and SHOWDEF.

If *LISTNAME* is non-NIL, this means that you specified *LISTNAME* as the filevar in interaction with ADDTOFILES? (see the FileVars section).

If no NEWCOM is specified, the default is to call DEFAULTMAKENEWCOM, which will construct and return a command of the form (*TYPE NAME*). You can advise or redefine DEFAULTMAKENEWCOM.

WHENCHANGED [File Manager Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *REASON* when *NAME*, an instance of type *TYPE*, is changed or defined (see MARKASCHANGED, in the Marking Changes section). Used for various applications, e.g. when an object of type I.S.OPRS changes, it is necessary to clear the corresponding translatons from CLISPARRAY.

The WHENCHANGED functions are called before the object is marked as changed, so that it can, in fact, decide that the object is *not* to be marked as changed, and execute (RETFROM 'MARKASCHANGED).

The *REASON* argument passed to WHENCHANGED functions is either DEFINED or CHANGED.

WHENFILED [File Manager Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is added to *FILE*.

WHENUNFILED [File Manager Type Property]

Value is a list of functions to be applied to *NAME*, *TYPE*, and *FILE* when *NAME*, an instance of type *TYPE*, is removed from *FILE*.

DESCRIPTION [File Manager Type Property]

Value is a string which describes instances of this type. For example, for type RECORDS, the value of DESCRIPTION is the string "record declarations".

The function FILEPKGTYPE is used to define new file manager types, or to change the properties of existing types. It is possible to redefine the attributes of system file manager types, such as FNS or PROPS.

(**FILEPKGTYPE** *TYPE PROP$_1$ VAL$_1$ ... PROP$_N$ VAL$_N$*) [NoSpread Function]

Nospread function for defining new file manager types, or changing properties of existing file manager types. *PROP$_i$* is one of the property names given above; *VAL$_i$* is the value to be given to that property. Returns *TYPE*.

(FILEPKGTYPE *TYPE PROP*) returns the value of the property *PROP*, without changing it.

(FILEPKGTYPE *TYPE*) returns a property list of all of the defined properties of *TYPE*, using the property names as keys.

Specifying *TYPE* as the symbol TYPE can be used to define one file manager type as a synonym of another. For example, (FILEPKGTYPE 'R 'TYPE 'RECORDS) defines R as a synonym for the file manager type RECORDS.

File Manager Commands

The basic mechanism for creating symbolic files is the function MAKEFILE (see the Storing Files section). For each file, the file manager has a data structure known as the "filecoms", which specifies what typed descriptions are contained in the file. A filecoms is a list of file manager commands, each of which specifies objects of a certain file manager type which should be dumped. For example, the filecoms

```
((FNS FOO)
  (VARS FOO BAR BAZ)
  (RECORDS XYZZY))
```

has a `FNS`, a `VARS`, and a `RECORDS` file manager command. This filecoms specifies that the function definition for `FOO`, the variable values of `FOO`, `BAR`, and `BAZ`, and the record declaration for `XYZZY` should be dumped.

By convention, the filecoms of a file *X* is stored as the value of the symbol *X*COMS. For example, (`MAKEFILE 'FOO.;27`) will use the value of `FOOCOMS` as the filecoms. This variable can be directly manipulated, but the file manager contains facilities which make constructing and updating filecoms easier, and in some cases automatic (see the Functions for Manipulating File Command Lists section).

A file manager command is an instruction to `MAKEFILE` to perform an explicit, well-defined operation, usually printing an expression. Usually there is a one-to-one correspondence between file manager types and file manager commands; for each file manager type, there is a file manager command which is used for writing objects of that type to a file, and each file manager command is used to write objects of a particular type. However, in some cases, the same file manager type can be dumped by several different file manager commands. For example, the file manager commands `PROP`, `IFPROP`, and `PROPS` all dump out objects with the file manager type `PROPS`. This means if you change an object of file manager type `PROPS` via `EDITP`, a typed-in call to `PUTPROP`, or via an explicit call to `MARKASCHANGED`, this object can be written out with any of the above three commands. Thus, when the file manager attempts to determine whether this typed object is contained on a particular file, it must look at instances of all three file manager commands `PROP`, `IFPROP`, and `PROPS`, to see if the corresponding atom and property are specified. It is also permissible for a single file manager command to dump several different file manager types. For example, you can define a file manager command which dumps both a function definition and its macro. Conversely, some file manager comands do not dump any file manager types at all, such as the `E` command.

For each file manager command, the file manager must be able to determine what typed definitions the command will cause to be printed so that the file manager can determine on what file (if any) an object of a given type is contained (by searching through the filecoms). Similarly, for each file manager type, the file manager must be able to construct a command that will print out an object of that type. In other words, the file manager must be able to map file manager commands into file manager types, and vice versa. Information can be provided to the file manager about a particular file manager command via the function `FILEPKGCOM` (see the Defining New File Manager Commands section), and information about a particular file manager type via the function `FILEPKGTYPE` (see the prior section). In the absence of other information, the default is simply that a file manager command of the form (*X NAME*) prints out the definition of *NAME* as a type *X*, and, conversely, if *NAME* is an object of type *X*, then *NAME* can be written out by a command of the form (*X NAME*).

If a file manager function is given a command or type that is not defined, it attempts spelling correction using `FILEPKGCOMSPLST` as a spelling list (unless `DWIMFLG` or `NOSPELLFLG` = `NIL`; see Chapter 20). If successful, the corrected version of the list of file manager commands is written (again) on the output file, since at this point, the uncorrected list of file manager commands would already have been printed on the output file. When the file is loaded, this will result in *FILE*COMS being reset, and may cause a message to be printed, e.g., (`FOOCOMS RESET`). The value of `FOOCOMS` would then be the corrected version. If the spelling correction is unsuccessful, the file manager functions generate an error, `BAD FILE PACKAGE COMMAND`.

File package commands can be used to save on the output file definitions of functions, values of variables, property lists of atoms, advised functions, edit macros, record declarations, etc. The interpretation of each file manager command is documented in the following sections.

(**USERMACROS** $SYMBOL_1$ ... $SYMBOL_N$)　　　　　　　　　　　[File Manager Command]

> Each symbol $SYMBOL_i$ is the name of a user edit macro. Writes expressions to add the edit macro definitions of $SYMBOL_i$ to USERMACROS, and adds the names of the commands to the appropriate spelling lists.

> If $SYMBOL_i$ is not a user macro, a warning message "no EDIT MACRO for $SYMBOL_i$" is printed.

Functions and Macros

(**FNS** FN_1 ... FN_N)　　　　　　　　　　　　　　　　[File Manager Command]

> Writes a DEFINEQ expression with the function definitions of FN_1 ... FN_N.

> You should never print a DEFINEQ expression directly onto a file (by using the P file manager command, for example), because MAKEFILE generates the filemap of function definitions from the FNS file manager commands (see the File Maps section).

(**ADVISE** FN_1 ... FN_N)　　　　　　　　　　　　　　[File Manager Command]

> For each function FN_i, writes expressions to reinstate the function to its advised state when the file is loaded. See Chapter 15.

> When advice is applied to a function programmatically or by hand, it is additive. That is, if a function already has some advice, further advice is added to the already-existing advice. However, when advice is applied to a function as a result of loading a file with an ADVISE file manager command, the new advice replaces any earlier advice. ADVISE works this way to prevent problems with loading different versions of the same advice. If you really want to apply additive advice, a file manager command such as (P (ADVISE ...)) should be used (see the Miscellaneous File Manager Commands section).

(**ADVICE** FN_1 ... FN_N)　　　　　　　　　　　　　　[File Manager Command]

> For each function FN_i, writes a PUTPROPS expression which will put the advice back on the property list of the function. You can then use READVISE (see Chapter 15) to reactivate the advice.

(**MACROS** $SYMBOL_1$ ... $SYMBOL_N$)　　　　　　　　　　　[File Manager Command]

> Each $SYMBOL_i$ is a symbol with a MACRO definition (and/or a DMACRO, 10MACRO, etc.). Writes out an expression to restore all of the macro properties for each $SYMBOL_i$, embedded in a DECLARE: EVAL@COMPILE so the macros will be defined when the file is compiled. See Chapter 10.

Variables

(**VARS** VAR_1 ... VAR_N)   [File Manager Command]

> For each VAR_i, writes an expression to set its top level value when the file is loaded. If VAR_i is atomic, VARS writes out an expression to set VAR_i to the top-level value it had at the time the file was written. If VAR_i is non-atomic, it is interpreted as (VAR FORM), and VARS write out an expression to set VAR to the value of FORM (evaluated when the file is loaded).

> VARS prints out expressions using RPAQQ and RPAQ, which are like SETQQ and SETQ except that they also perform some special operations with respect to the file manager (see the Functions Used within Source Files section).

> VARS cannot be used for putting arbitrary variable values on files. For example, if the value of a variable is an array (or many other data types), a symbol which represents the array is dumped in the file instead of the array itself. The HORRIBLEVARS file manager command provides a way of saving and reloading variables whose values contain re-entrant or circular list structure, user data types, arrays, or hash arrays.

(**INITVARS** VAR_1 ... VAR_N)   [File Manager Command]

> INITVARS is used for initializing variables, setting their values only when they are currently NOBIND. A variable value defined in an INITVARS command will not change an already established value. This means that re-loading files to get some other information will not automatically revert to the initialization values.

> The format of an INITVARS command is just like VARS. The only difference is that if VAR_i is atomic, the current value is not dumped; instead NIL is defined as the initialization value. Therefore, (INITVARS FOO (FUM 2)) is the same as (VARS (FOO NIL)(FUM 2)), if FOO and FUM are both NOBIND.

> INITVARS writes out an RPAQ? expression on the file instead of RPAQ or RPAQQ.

(**ADDVARS** (VAR_1 . LST_1)...(VAR_N . LST_N))   [File Manager Command]

> For each (VAR_i . LST_i), writes an ADDTOVAR (see the Functions Used Within Source Files section) to add each element of LST_i to the list that is the value of VAR_i at the time the file is loaded. The new value of VAR_i will be the union of its old value and LST_i. If the value of VAR_i is NOBIND, it is first set to NIL.

> For example, (ADDVARS (DIRECTORIES LISP LISPUSERS)) will add LISP and LISPUSERS to the value of DIRECTORIES.

> If LST_i is not specified, VAR_i is initialized to NIL if its current value is NOBIND. In other words, (ADDVARS (VAR)) will initialize VAR to NIL if VAR has not previously been set.

(**APPENDVARS** $(VAR_1 . LST_1)$ ... $(VAR_N . LST_N)$) [File Manager Command]

The same as ADDVARS, except that the values are added to the end of the lists (using APPENDTOVAR, in the Functions Used Within Source Files section), rather than at the beginning.

(**UGLYVARS** VAR_1 ... VAR_N) [File Manager Command]

Like VARS, except that the value of each VAR_i may contain structures for which READ is not an inverse of PRINT, e.g. arrays, readtables, user data types, etc. Uses HPRINT (see Chapter 25).

(**HORRIBLEVARS** VAR_1 ... VAR_N) [File Manager Command]

Like UGLYVARS, except structures may also contain circular pointers. Uses HPRINT (see Chapter 25). The values of VAR_1 ... VAR_N are printed in the same operation, so that they may contain pointers to common substructures.

UGLYVARS does not do any checking for circularities, which results in a large speed and internal-storage advantage over HORRIBLEVARS. Thus, if it is known that the data structures do *not* contain circular pointers, UGLYVARS should be used instead of HORRIBLEVARS.

(**ALISTS** $(VAR_1 KEY_1 KEY_2 ...)...(VAR_N KEY_3 KEY_4 ...)$) [File Manager Command]

VAR_i is a variable whose value is an association list, such as EDITMACROS, BAKTRACELST, etc. For each VAR_i, ALISTS writes out expressions which will restore the values associated with the specified keys. For example, (ALISTS (BREAKMACROS BT BTV)) will dump the definition for the BT and BTV commands on BREAKMACROS.

Some association lists (USERMACROS, LISPXMACROS, etc.) are used to implement other file manager types, and they have their own file manager commands.

(**SPECVARS** VAR_1 ... VAR_N) [File Manager Command]
(**LOCALVARS** VAR_1 ... VAR_N) [File Manager Command]
(**GLOBALVARS** VAR_1 ... VAR_N) [File Manager Command]

Outputs the corresponding compiler declaration embedded in a DECLARE: DOEVAL@COMPILE DONTCOPY. See Chapter 18.

(**CONSTANTS** VAR_1 ... VAR_N) [File Manager Command]

Like VARS, for each VAR_i writes an expression to set its top level value when the file is loaded. Also writes a CONSTANTS expression to declare these variables as constants (see Chapter 18). Both of these expressions are wrapped in a (DECLARE: EVAL@COMPILE ...) expression, so they can be used by the compiler.

Like VARS, VAR_i can be non-atomic, in which case it is interpreted as $(VAR\ FORM)$, and passed to CONSTANTS (along with the variable being initialized to $FORM$).

Symbol Properties

($\textbf{PROP}$ $PROPNAME$ $SYMBOL_1$ ... $SYMBOL_N$)   [File Manager Command]

Writes a PUTPROPS expression to restore the value of the $PROPNAME$ property of each symbol $SYMBOL_i$ when the file is loaded.

If $PROPNAME$ is a list, expressions will be written for each property on that list. If $PROPNAME$ is the symbol ALL, the values of all user properties (on the property list of each $SYMBOL_i$) are saved. SYSPROPS is a list of properties used by system functions. Only properties *not* on that list are dumped when the ALL option is used.

If $SYMBOL_i$ does not have the property $PROPNAME$ (as opposed to having the property with value NIL), a warning message "NO $PROPNAME$ PROPERTY FOR $SYMBOL_i$" is printed. The command IFPROP can be used if it is not known whether or not an atom will have the corresponding property.

($\textbf{IFPROP}$ $PROPNAME$ $SYMBOL_1$ ... $SYMBOL_N$)   [File Manager Command]

Same as the PROP file manager command, except that it only saves the properties that actually appear on the property list of the corresponding atom. For example, if FOO1 has property PROP1 and PROP2, FOO2 has PROP3, and FOO3 has property PROP1 and PROP3, then (IFPROP (PROP1 PROP2 PROP3) FOO1 FOO2 FOO3) will save only those five property values.

($\textbf{PROPS}$ $(SYMBOL_1\ PROPNAME_1)...(SYMBOL_N\ PROPNAME_N)$)   [File Manager Command]

Similar to PROP command. Writes a PUTPROPS expression to restore the value of $PROPNAME_i$ for each $SYMBOL_i$ when the file is loaded.

As with the PROP command, if $SYMBOL_i$ does not have the property $PROPNAME$ (as opposed to having the property with NIL value), a warning message "NO $PROPNAME_i$ PROPERTY FOR $SYMBOL_i$" is printed.

Miscellaneous File Manager Commands

($\textbf{RECORDS}$ REC_1 ... REC_N)   [File Manager Command]

Each REC_i is the name of a record (see Chapter 8). Writes expressions which will redeclare the records when the file is loaded.

($\textbf{INITRECORDS}$ REC_1 ... REC_N)   [File Manager Command]

Similar to RECORDS, INITRECORDS writes expressions on a file that will, when loaded, perform whatever initialization/allocation is necessary for the indicated records.

However, the record declarations themselves are not written out. This facility is useful for building systems on top of Interlisp, in which the implementor may want to eliminate the record declarations from a production version of the system, but the allocation for these records must still be done.

(**LISPXMACROS** $SYMBOL_1$ ... $SYMBOL_N$) [File Manager Command]

Each $SYMBOL_i$ is defined on LISPXMACROS or LISPXHISTORYMACROS (see Chapter 13). Writes expressions which will save and restore the definition for each macro, as well as making the necessary additions to LISPXCOMS

(**I.S.OPRS** OPR_1 ... OPR_N) [File Manager Command]

Each OPR_i is the name of a user-defined i.s.opr (see Chapter 9). Writes expressions which will redefine the i.s.oprs when the file is loaded.

(**RESOURCES** $RESOURCE_1$ ... $RESOURCE_N$) [File Manager Command]

Each $RESOURCES_i$ is the name of a resource (see Chapter 12). Writes expressions which will redeclare the resource when the file is loaded.

(**INITRESOURCES** $RESOURCE_1$ ... $RESOURCE_N$) [File Manager Command]

Parallel to INITRECORDS, INITRESOURCES writes expressions on a file to perform whatever initialization/allocation is necessary for the indicated resources, without writing the resource declaration itself.

(**COURIERPROGRAMS** $NAME_1$ ... $NAME_N$) [File Manager Command]

Each $NAME_i$ is the name of a Courier program (see Chapter 31). Writes expressions which will redeclare the Courier program when the file is loaded.

(**TEMPLATES** $SYMBOL_1$ ... $SYMBOL_N$) [File Manager Command]

Each $SYMBOL_i$ is a symbol which has a Masterscope template (see Chapter 19). Writes expressions which will restore the templates when the file is loaded.

(**FILES** $FILE_1$ ... $FILE_N$) [File Manager Command]

Used to specify auxiliary files to be loaded in when the file is loaded. Dumps an expression calling FILESLOAD (see the Loading Files section), with $FILE_1$ ... $FILE_N$ as the arguments. FILESLOAD interprets $FILE_1$ ... $FILE_N$ as files to load, possibly interspersed with lists used to specify certain loading options.

(**FILEPKGCOMS** $SYMBOL_1$ ... $SYMBOL_N$) [File Manager Command]

Each symbol $SYMBOL_i$ is either the name of a user-defined file manager command or a user-defined file manager type (or both). Writes expressions which will restore each command/type.

If $SYMBOL_i$ is not a file manager command or type, a warning message `"no FILE PACKAGE COMMAND for SYMBOL_i"` is printed.

(*** .** *TEXT*)   [File Manager Command]

Used for inserting comments in a file. The file manager command is simply written on the output file; it will be ignored when the file is loaded.

If the first element of *TEXT* is another `*`, a form-feed is printed on the file before the comment.

(**P** EXP_1 ... EXP_N)   [File Manager Command]

Writes each of the expressions EXP_1 ... EXP_N on the output file, where they will be evaluated when the file is loaded.

(**E** $FORM_1$ ... $FORM_N$)   [File Manager Command]

Each of the forms $FORM_1$ ... $FORM_N$ is evaluated at *output* time, when MAKEFILE interprets this file manager command.

(**COMS** COM_1 ... COM_N)   [File Manager Command]

Each of the commands COM_1 ... COM_N is interpreted as a file manager command.

(**ORIGINAL** COM_1 ... COM_N)   [File Manager Command]

Each of the commands COM_i will be interpreted as a file manager command without regard to any file manager macros (as defined by the MACRO property of the FILEPKGCOM function, in the Defining New File Manager Commands section). Useful for redefining a built-in file manager command in terms of itself.

Some of the "built-in" file manager commands are defined by file manager macros, so interpreting them (or new user-defined file manager commands) with ORIGINAL will fail. ORIGINAL was never intended to be used outside of a file manager command macro.

DECLARE:

(**DECLARE:** . *FILEPKGCOMS/FLAGS*)   [File Manager Command]

Normally expressions written onto a symbolic file are evaluated when loaded; copied to the compiled file when the symbolic file is compiled (see Chapter 18); and not evaluated at compile time. DECLARE: allows you to override these defaults.

FILEPKGCOMS/FLAGS is a list of file manager commands, possibly interspersed with "tags". The output of those file manager commands within *FILEPKGCOMS/FLAGS* is embedded in a DECLARE: expression, along with any tags that are specified. For example, (DECLARE: EVAL@COMPILE DONTCOPY (FNS ...) (PROP ...)) would produce (DECLARE: EVAL@COMPILE DONTCOPY (DEFINEQ ...) (PUTPROPS

`...))`.   `DECLARE:` is *defined* as an nlambda nospread function, which processes its arguments by evaluating or not evaluating each expression depending on the setting of internal state variables. The initial setting is to evaluate, but this can be overridden by specifying the `DONTEVAL@LOAD` tag.

`DECLARE:` expressions are specially processed by the compiler. For the purposes of compilation, `DECLARE:` has two principal applications: to specify forms that are to be evaluated at compile time, presumably to affect the compilation, e.g., to set up macros; and/or to indicate which expressions appearing in the symbolic file are *not* to be copied to the output file. (Normally, expressions are *not* evaluated and *are* copied.) Each expression in `CDR` of a `DECLARE:` form is either evaluated/not-evaluated and copied/not-copied depending on the settings of two internal state variables, initially set for copy and not-evaluate. These state variables can be reset for the remainder of the expressions in the `DECLARE:` by means of the tags `DONTCOPY`, `EVAL@COMPILE`, etc.

The tags are:

EVAL@LOAD

DOEVAL@LOAD   Evaluate the following forms when the file is loaded (unless overridden by `DONTEVAL@LOAD`).

DONTEVAL@LOAD   Do not evaluate the following forms when the file is loaded.

EVAL@LOADWHEN   This tag can be used to provide conditional evaluation. The value of the expression immediately following the tag determines whether or not to evaluate subsequent expressions when loading. `...` `EVAL@LOADWHEN` `T` `...` is equivalent to `...` `EVAL@LOAD` `...`

COPY

DOCOPY   When compiling, copy the following forms into the compiled file.

DONTCOPY   When compiling, do not copy the following forms into the compiled file.

Note: If the file manager commands following `DONTCOPY` include record declarations for datatypes, or records with initialization forms, it is necessary to include a `INITRECORDS` file manager command (see the prior section) outside of the `DONTCOPY` form so that the initialization information is copied. For example, if `FOO` was defined as a datatype,

```
(DECLARE: DONTCOPY (RECORDS FOO))
(INITRECORDS FOO)
```

would copy the data type declaration for `FOO`, but would not copy the whole record declaration.

COPYWHEN	When compiling, if the next form evaluates to non-`NIL`, copy the following forms into the compiled file.
EVAL@COMPILE	
DOEVAL@COMPILE	When compiling, evaluate the following forms.
DONTEVAL@COMPILE	When compiling, do not evaluate the following forms.
EVAL@COMPILEWHEN	When compiling, if the next form evaluates to non-`NIL`, evaluate the following forms.
FIRST	For expressions that are to be copied to the compiled file, the tag `FIRST` can be used to specify that the following expressions in the `DECLARE:` are to appear at the front of the compiled file, before anything else except the `FILECREATED` expressions (see the Symbolic File Format section). For example, (`DECLARE: COPY FIRST (P (PRINT` *MESS1* `T)) NOTFIRST (P (PRINT` *MESS2* `T)))` will cause (`PRINT` *MESS1* `T`) to appear first in the compiled file, followed by any functions, then (`PRINT MESS2 T`).
NOTFIRST	Reverses the effect of `FIRST`.

The value of `DECLARETAGSLST` is a list of all the tags used in `DECLARE:` expressions. If a tag not on this list appears in a `DECLARE:` file manager command, spelling correction is performed using `DECLARETAGSLST` as a spelling list.

Note that the function `LOADCOMP` (see the Loading Files section) provides a convenient way of obtaining information from the `DECLARE:` expressions in a file, without reading in the entire file. This information may be used for compiling other files.

(**BLOCKS** *BLOCK$_1$ ... BLOCK$_N$*)   [File Manager Command]

For each *BLOCK$_i$*, writes a `DECLARE:` expression which the block compile functions interpret as a block declaration. See Chapter 18.

Exporting Definitions

When building a large system in Interlisp, it is often the case that there are record definitions, macros and the like that are needed by several different system files when running, analyzing and compiling the source code of the system, but which are not needed for running the compiled code. By using the `DECLARE:` file manager command with tag `DONTCOPY` (see the prior section), these definitions can be kept out of the compiled files, and hence out of the system constructed by loading the compiled files files into Interlisp. This saves loading time, space in the resulting system, and whatever other overhead might be incurred by keeping those definitions around, e.g., burden on the record package to consider more possibilities in translating record accesses, or conflicts between system record fields and user record fields.

However, if the implementor wants to debug or compile code in the resulting system, the definitions are needed. And even if the definitions *had* been copied to the compiled files, a similar problem arises if one wants to work on system code in a regular Interlisp environment where none of the system files had been loaded. One could mandate that any definition needed by more than one file in the system should reside on a distinguished file of definitions, to be loaded into any environment where the system files are worked on. Unfortunately, this would keep the definitions away from where they logically belong. The EXPORT mechanism is designed to solve this problem.

To use the mechanism, the implementor identifies any definitions needed by files other than the one in which the definitions reside, and wraps the corresponding file manager commands in the EXPORT file manager command. Thereafter, GATHEREXPORTS can be used to make a single file containing all the exports.

(**EXPORT** COM_1 ... COM_N)   [File Manager Command]

> This command is used for "exporting" definitions. Like COM, each of the commands COM_1 ... COM_N is interpreted as a file manager command. The commands are also flagged in the file as being "exported" commands, for use with GATHEREXPORTS.

(**GATHEREXPORTS** *FROMFILES TOFILE FLG*)   [Function]

> *FROMFILES* is a list of files containing EXPORT commands. GATHEREXPORTS extracts all the exported commands from those files and produces a loadable file *TOFILE* containing them. If *FLG* = EVAL, the expressions are evaluated as they are gathered; i.e., the exports are effectively loaded into the current environment as well as being written to *TOFILE*.

(**IMPORTFILE** *FILE RETURNFLG*)   [Function]

> If *RETURNFLG* is NIL, this loads any exported definitions from *FILE* into the current environment. If *RETURNFLG* is T, this returns a list of the exported definitions (evaluable expressions) without actually evaluating them.

(**CHECKIMPORTS** *FILES NOASKFLG*)   [Function]

> Checks each of the files in *FILES* to see if any exists in a version newer than the one from which the exports in memory were taken (GATHEREXPORTS and IMPORTFILE note the creation dates of the files involved), or if any file in the list has not had its exports loaded at all. If there are any such files, you are asked for permission to IMPORTFILE each such file. If *NOASKFLG* is non-NIL, IMPORTFILE is performed without asking.

For example, suppose file FOO contains records R1, R2, and R3, macros BAR and BAZ, and constants CON1 and CON2. If the definitions of R1, R2, BAR, and BAZ are needed by files other than FOO, then the file commands for FOO might contain the command

```
(DECLARE: EVAL@COMPILE DONTCOPY
   (EXPORT (RECORDS R1 R2)
           (MACROS BAR BAZ))
   (RECORDS R3)
   (CONSTANTS BAZ))
```

None of the commands inside this `DECLARE:` would appear on `FOO`'s compiled file, but `(GATHEREXPORTS '(FOO) 'MYEXPORTS)` would copy the record definitions for `R1` and `R2` and the macro definitions for `BAR` and `BAZ` to the file `MYEXPORTS`.

FileVars

In each of the file manager commands described above, if the symbol `*` follows the command type, the form following the `*`, i.e., `CADDR` of the command, is evaluated and its value used in executing the command, e.g., `(FNS * (APPEND FNS1 FNS2))`. When this form is a symbol, e.g. `(FNS * FOOFNS)`, we say that the variable is a "filevar". Note that `(COMS * FORM)` provides a way of *computing* what should be done by `MAKEFILE`.

Example:

```
← (SETQ FOOFNS '(FOO1 FOO2 FOO3))
        (FOO1 FOO2 FOO3)

← (SETQ FOOCOMS
        '((FNS * FOOFNS)
        (VARS FIE)
        (PROP MACRO FOO1 FOO2)
        (P (MOVD 'FOO1 'FIE1))]

← (MAKEFILE 'FOO)
```

would create a file `FOO` containing:

```
(FILECREATED  "time  and  date  the  file  was  made"  .  "other
information")
        (PRETTYCOMPRINT FOOCOMS)
        (RPAQQ FOOCOMS ((FNS * FOOFNS) ...))
        (RPAQQ FOOFNS (FOO1 FOO3 FOO3))
        (DEFINEQ "definitions of FOO1, FOO2, and FOO3")
        (RPAQQ FIE "value of FIE")
        (PUTPROPS FOO1 MACRO PROPVALUE)
        (PUTPROPS FOO2 MACRO PROPVALUE)
        (MOVD (QUOTE FOO1) (QUOTE FIE1))
        STOP
```

For the `PROP` and `IFPROP` commands (see the Litatom Properties section), the `*` follows the property name instead of the command, e.g., `(PROP MACRO * FOOMACROS)`. Also, in the form `(* * comment ...)`, the word `comment` is not treated as a filevar.

Defining New File Manager Commands

A file manager command is defined by specifying the values of certain properties. You can specify the various attributes of a file manager command for a new command, or respecify them for an existing command. The following properties are used:

MACRO   [File Manager Command Property]

Defines how to dump the file manager command. Used by `MAKEFILE`. Value is a pair (`ARGS` . `COMS`). The "arguments" to the file manager command are substituted for `ARGS` throughout `COMS`, and the result treated as a list of file manager commands. For example, following (`FILEPKGCOM` `'FOO` `'MACRO` `'((X Y) . COMS)`), the file manager command (`FOO A B`) will cause `A` to be substituted for `X` and `B` for `Y` throughout `COMS`, and then `COMS` treated as a list of commands.

The substitution is carried out by `SUBPAIR` (see Chapter 3), so that the "argument list" for the macro can also be atomic. For example, if (`X` . `COMS`) was used instead of (`(X Y)` . `COMS`), then the command (`FOO A B`) would cause (`A B`) to be substituted for `X` throughout `COMS`.

Filevars are evaluated *before* substitution. For example, if the symbol `*` follows `NAME` in the command, `CADDR` of the command is evaluated substituting in `COMS`.

ADD   [File Manager Command Property]

Specifies how (if possible) to add an instance of an object of a particular type to a given file manager command. Used by `ADDTOFILE`. Value is `FN`, a function of three arguments, `COM`, a file manager command `CAR` of which is `EQ` to `COMMANDNAME`, `NAME`, a typed object, and `TYPE`, its type. `FN` should return `T` if it (undoably) adds `NAME` to `COM`, `NIL` if not. If no `ADD` property is specified, then the default is (1) if (`CAR COM`) = `TYPE` and (`CADR COM`) = `*`, and (`CADDR COM`) is a filevar (i.e. a literal atom), add `NAME` to the value of the filevar, or (2) if (`CAR COM`) = `TYPE` and (`CADR COM`) is not `*`, add `NAME` to (`CDR COM`).

Actually, the function is given a fourth argument, `NEAR`, which if non-`NIL`, means the function should try to add the item after `NEAR`. See discussion of `ADDTOFILES?`, in the Storing Files section.

DELETE   [File Manager Command Property]

Specifies how (if possible) to delete an instance of an object of a particular type from a given file manager command. Used by `DELFROMFILES`. Value is `FN`, a function of three arguments, `COM`, `NAME`, and `TYPE`, same as for `ADD`. `FN` should return `T` if it (undoably) deletes `NAME` from `COM`, `NIL` if not. If no `DELETE` property is specified, then the default is either (`CAR COM`) = `TYPE` and (`CADR COM`) = `*`, and (`CADDR COM`) is a filevar (i.e. a literal atom), and `NAME` is contained in the value of the filevar, then remove `NAME` from the filevar, or if (`CAR COM`) = `TYPE` and (`CADR COM`) is not `*`, and `NAME` is contained in (`CDR COM`), then remove `NAME` from (`CDR COM`).

If `FN` returns the value of `ALL`, it means that the command is now "empty", and can be deleted entirely from the command list.

CONTENTS [File Manager Command Property]

CONTAIN [File Manager Command Property]

Determines whether an instance of an object of a given type is contained in a given file manager command. Used by WHEREIS and INFILECOMS?. Value is *FN*, a function of three arguments, *COM*, a file manager command CAR of which is EQ to *COMMANDNAME*, *NAME*, and *TYPE*. The interpretation of *NAME* is as follows: if *NAME* is NIL, *FN* should return a list of elements of type *TYPE* contained in *COM*. If *NAME* is T, *FN* should return T if there are *any* elements of type *TYPE* in *COM*. If *NAME* is an atom other than T or NIL, return T if *NAME* of type *TYPE* is contained in *COM*. Finally, if *NAME* is a list, return a list of those elements of type *TYPE* contained in *COM* that are also contained in *NAME*.

It is sufficient for the CONTENTS function to simply return the list of items of type *TYPE* in command *COM*, i.e. it can in fact ignore the *NAME* argument. The *NAME* argument is supplied mainly for those situations where producing the entire list of items involves significantly more computation or creates more storage than simply determining whether a particular item (or any item) of type *TYPE* is contained in the command.

If a CONTENTS property is specified and the corresponding function application returns NIL and (CAR *COM*) = *TYPE*, then the operation indicated by *NAME* is performed on the value of (CADDR *COM*), if (CADR *COM*) = *, otherwise on (CDR *COM*). In other words, by specifying a CONTENTS property that returns NIL, e.g. the function NILL, you specify that a file manager command of name FOO produces objects of file manager type FOO and only objects of type FOO.

If the CONTENTS property is not provided, the command is simply expanded according to its MACRO definition, and each command on the resulting command list is then interrogated.

If *COMMANDNAME* is a file manager command that is used frequently, its expansion by the various parts of the system that need to interrogate files can result in a large number of CONSes and garbage collections. By informing the file manager as to what this command actually does and does not produce via the CONTENTS property, this expansion is avoided. For example, suppose you have a file manager command called GRAMMARS which dumps various property lists but no functions. The file manager could ignore this command when seeking information about FNS.

The function FILEPKGCOM is used to define new file manager commands, or to change the properties of existing commands. It is possible to redefine the attributes of system file manager commands, such as FNS or PROPS, and to cause unpredictable results.

(**FILEPKGCOM** *COMMANDNAME PROP$_1$ VAL$_1$ ... PROP$_N$ VAL$_N$*) [NoSpread Function]

Nospread function for defining new file manager commands, or changing properties of existing file manager commands. *PROP$_i$* is one of of the property names described above; *VAL$_i$* is the value to be given that property of the file manager command *COMMANDNAME*. Returns *COMMANDNAME*.

(FILEPKGCOM *COMMANDNAME* *PROP*) returns the value of the property *PROP*, without changing it.

(FILEPKGCOM *COMMANDNAME*) returns a property list of all of the defined properties of *COMMANDNAME*, using the property names as keys.

Specifying *TYPE* as the symbol COM can be used to define one file manager command as a synonym of another. For example, (FILEPKGCOM 'INITVARIABLES 'COM 'INITVARS) defines INITVARIABLES as a synonym for the file manager command INITVARS.

Functions for Manipulating File Command Lists

The following functions may be used to manipulate filecoms. The argument *COMS* does *not* have to correspond to the filecoms for some file. For example, *COMS* can be the list of commands generated as a result of expanding a user-defined file manager command.

The following functions will accept a file manager command as a valid value for their *TYPE* argument, even if it does not have a corresponding file manager type. User-defined file manager commands are expanded as necessary.

(**INFILECOMS**? *NAME TYPE COMS*)   [Function]

COMS is a list of file manager commands, or a variable whose value is a list of file manager commands. *TYPE* is a file manager type. INFILECOMS? returns T if *NAME* of type *TYPE* is "contained" in *COMS*.

If *NAME* = NIL, INFILECOMS? returns a list of all elements of type *TYPE*.

If *NAME* = T, INFILECOMS? returns T if there are *any* elements of type *TYPE* in *COMS*.

(**ADDTOFILE** *NAME TYPE FILE NEAR LISTNAME*)   [Function]

Adds *NAME* of type *TYPE* to the file manager commands for *FILE*. If *NEAR* is given and it is the name of an item of type *TYPE* already on *FILE*, then *NAME* is added to the command that dumps *NEAR*. If *LISTNAME* is given and is the name of a list of items of *TYPE* items on *FILE*, then *NAME* is added to that list. Uses ADDTOCOMS and MAKENEWCOM. Returns *FILE*. ADDTOFILE is undoable.

(**DELFROMFILES** *NAME TYPE FILES*)   [Function]

Deletes all instances of *NAME* of type *TYPE* from the filecoms for each of the files on *FILES*. If *FILES* is a non-NIL symbol, (LIST *FILES*) is used. *FILES* = NIL defaults to FILELST. Returns a list of files from which *NAME* was actually removed. Uses DELFROMCOMS. DELFROMFILES is undoable.

Deleting a function will also remove the function from any BLOCKS declarations in the filecoms.

(**ADDTOCOMS** *COMS NAME TYPE NEAR LISTNAME*) [Function]

Adds *NAME* as a *TYPE* to *COMS*, a list of file manager commands or a variable whose value is a list of file manager commands. Returns NIL if ADDTOCOMS was unable to find a command appropriate for adding *NAME* to *COMS*. *NEAR* and *LISTNAME* are described in the discussion of ADDTOFILE. ADDTOCOMS is undoable.

The exact algorithm for adding commands depends the particular command itself. See discussion of the ADD property, in the description of FILEPKGCOM.

ADDTOCOMS will not attempt to add an item to any command which is inside of a DECLARE: unless you specified a specific name via the LISTNAME or NEAR option of ADDTOFILES?.

(**DELFROMCOMS** *COMS NAME TYPE*) [Function]

Deletes *NAME* as a *TYPE* from *COMS*. Returns NIL if DELFROMCOMS was unable to modify *COMS* to delete *NAME*. DELFROMCOMS is undoable.

(**MAKENEWCOM** *NAME TYPE*) [Function]

Returns a file manager command for dumping *NAME* of type *TYPE*. Uses the procedure described in the discussion of NEWCOM, in the Defining New File Manager Types section.

(**MOVETOFILE** *TOFILE NAME TYPE FROMFILE*) [Function]

Moves the definition of *NAME* as a *TYPE* from *FROMFILE* to *TOFILE* by modifying the file commands in the appropriate way (with DELFROMFILES and ADDTOFILE).

Note that if *FROMFILE* is specified, the definition will be retrieved from that file, even if there is another definition currently in your environment.

(**FILECOMSLST** *FILE TYPE*) [Function]

Returns a list of all objects of type *TYPE* in *FILE*.

(**FILEFNSLST** *FILE*) [Function]

Same as (FILECOMSLST *FILE* 'FNS).

(**FILECOMS** *FILE TYPE*) [Function]

Returns (PACK* *FILE* (OR *TYPE* 'COMS)). Note that (FILECOMS 'FOO) returns the symbol FOOCOMS, not the value of FOOCOMS.

(**SMASHFILECOMS** *FILE*) [Function]

Maps down (FILECOMSLST *FILE* 'FILEVARS) and sets to NOBIND all filevars (see the FileVars section), i.e., any variable used in a command of the form (*COMMAND* * *VARIABLE*). Also sets (FILECOMS *FILE*) to NOBIND. Returns *FILE*.

Symbolic File Format

The file manager manipulates symbolic files in a particular format. This format is defined so that the information in the file is easily readable when the file is listed, as well as being easily manipulated by the file manager functions. In general, there is no reason for you to manually change the contents of a symbolic file. However, to allow you to extend the file manager, this section describes some of the functions used to write symbolic files, and other matters related to their format.

(**PRETTYDEF** *PRTTYFNS PRTTYFILE PRTTYCOMS REPRINTFNS SOURCEFILE CHANGES*) [Function]

Writes a symbolic file in PRETTYPRINT format for loading, using FILERDTBL as its read table. PRETTYDEF returns the name of the symbolic file that was created.

PRETTYDEF operates under a RESETLST (see Chapter 14), so if an error occurs, or a Control-D is typed, all files that PRETTYDEF has opened will be closed, the (partially complete) file being written will be deleted, and any undoable operations executed will be undone. The RESETLST also means that any RESETSAVEs executed in the file manager commands will also be protected.

PRTTYFNS is an optional list of function names. It is equivalent to including (FNS * *PRTTYFNS*) in the file manager commands in *PRTTYCOMS*. *PRTTYFNS* is an anachronism from when PRETTYDEF did not use a list of file manager commands, and should be specified as NIL.

PRTTYFILE is the name of the file on which the output is to be written. *PRTTYFILE* has to be a symbnol. If *PRTTYFILE* = NIL, the primary output file is used. *PRTTYFILE* is opened if not already open, and it becomes the primary output file. *PRTTYFILE* is closed at end of PRETTYDEF, and the primary output file is restored.

PRTTYCOMS is a list of file manager commands interpreted as described in the File Manager Commands section. If *PRTTYCOMS* is atomic, its top level value is used and an RPAQQ is written which will set that atom to the list of commands when the file is subsequently loaded. A PRETTYCOMPRINT expression (see below) will also be written which informs you of the named atom or list of commands when the file is subsequently loaded. In addition, if any of the functions in the file are nlambda functions, PRETTYDEF will automatically print a DECLARE: expression suitable for informing the compiler about these functions, in case you recompile the file without having first loaded the nlambda functions (see Chapter 18).

REPRINTFNS and *SOURCEFILE* are for use in conjunction with remaking a file (see the Remaking a Symbolic File section). *REPRINTFNS* can be a list of functions to be prettyprinted, or EXPRS, meaning prettyprint all functions with EXPR definitions, or ALL meaning prettyprint all functions either defined as EXPRs, or with EXPR properties. Note that doing a remake with *REPRINTFNS* = NIL makes sense if there have been changes in the file, but not to any of the functions, e.g., changes to variables or property lists. *SOURCEFILE* is the name of the file from which to copy the definitions for those functions that are *not* going to be prettyprinted, i.e., those not specified by *REPRINTFNS*. *SOURCEFILE* = T means to use most recent version (i.e., highest number) of

PRTTYFILE, the second argument to PRETTYDEF. If *SOURCEFILE* cannot be found, PRETTYDEF prints the message "*FILE* NOT FOUND, SO IT WILL BE WRITTEN ANEW", and proceeds as it does when *REPRINTFNS* and *SOURCEFILE* are both NIL.

PRETTYDEF calls PRETTYPRINT with its second argument *PRETTYDEFLG* = T, so whenever PRETTYPRINT starts a new function, it prints (on the terminal) the name of that function if more than 30 seconds (real time) have elapsed since the last time it printed the name of a function.

Note that normally if PRETTYPRINT is given a symbol which is not defined as a function but is known to be on one of the files noticed by the file manager, PRETTYPRINT will load in the definition (using LOADFNS) and print it. This is not done when PRETTYPRINT is called from PRETTYDEF.

In Medley the SYSPRETTYFLG is ignored in the Interlisp exec.

(**PRINTFNS** *X*)   [Function]

X is a list of functions. PRINTFNS prettyprints a DEFINEQ epression that defines the functions to the primary output stream using the primary read table. Used by PRETTYDEF to implement the FNS file manager command.

(**PRINTDATE** *FILE CHANGES*)   [Function]

Prints the FILECREATED expression at beginning of PRETTYDEF files. *CHANGES* used by the file manager.

(**FILECREATED** *X*)   [NLambda NoSpread Function]

Prints a message (using LISPXPRINT) followed by the time and date the file was made, which is (CAR *X*). The message is the value of PRETTYHEADER, initially "FILE CREATED". If PRETTYHEADER = NIL, nothing is printed. (CDR *X*) contains information about the file, e.g., full name, address of file map, list of changed items, etc. FILECREATED also stores the time and date the file was made on the property list of the file under the property FILEDATES and performs other initialization for the file manager.

(**PRETTYCOMPRINT** *X*)   [NLambda Function]

Prints *X* (unevaluated) using LISPXPRINT, unless PRETTYHEADER = NIL.

PRETTYHEADER   [Variable]

Value is the message printed by FILECREATED. PRETTYHEADER is initially "FILE CREATED". If PRETTYHEADER = NIL, neither FILECREATED nor PRETTYCOMPRINT will print anything. Thus, setting PRETTYHEADER to NIL will result in "silent loads". PRETTYHEADER is reset to NIL during greeting (see Chapter 12).

(**FILECHANGES** *FILE TYPE*)   [Function]

> Returns a list of the changed objects of file manager type *TYPE* from the FILECREATED expression of *FILE*. If *TYPE* = NIL, returns an alist of all of the changes, with the file manager types as the CARs of the elements..

(**FILEDATE** *FILE*)   [Function]

> Returns the file date contained in the FILECREATED expression of *FILE*.

(**LISPSOURCEFILEP** *FILE*)   [Function]

> Returns a non-NIL value if *FILE* is in file manager format and has a file map, NIL otherwise.

Copyright Notices

The system has a facility for automatically printing a copyright notice near the front of files, right after the FILECREATED expression, specifying the years it was edited and the copyright owner. The format of the copyright notice is:

```
(* Copyright (c) 1981 by Foo Bars Corporation)
```

Once a file has a copyright notice then every version will have a new copyright notice inserted into the file without your intervention. (The copyright information necessary to keep the copyright up to date is stored at the end of the file.).

Any year the file has been edited is considered a "copyright year" and therefore kept with the copyright information. For example, if a file has been edited in 1981, 1982, and 1984, then the copyright notice would look like:

```
(* Copyright (c) 1981,1982,1984 by Foo Bars Corporation)
```

When a file is made, if it has no copyright information, the system will ask you to specify the copyright owner (if COPYRIGHTFLG = T). You may specify one of the names from COPYRIGHTOWNERS, or give one of the following responses:

- Type a left-square-bracket. The system will then prompt for an arbitrary string which will be used as the owner-string

- Type a right-square-bracket, which specifies that you really do not want a copyright notice.

- Type "NONE" which specifies that this file should never have a copyright notice.

For example, if COPYRIGHTOWNERS has the value

```
((BBN "Bolt Beranek and Newman Inc.")
 (XEROX "Xerox Corporation"))
```

then for a new file `FOO` the following interaction will take place:

```
Do you want to Copyright FOO? Yes
Copyright owner:   (user typed ?)
one of:
BBN - Bolt Beranek and Newman Inc.
XEROX - Xerox Corporation
NONE - no copyright ever for this file
[ - new copyright owner -- type one line of text
] - no copyright notice for this file now

Copyright owner: BBN
```

Then "Foo Bars Corporation" in the above copyright notice example would have been "Bolt Beranek and Newman Inc."

The following variables control the operation of the copyright facility:

COPYRIGHTFLG   [Variable]

> The value of COPYRIGHTFLG determines whether copyright information is maintained in files. Its value is interpreted as follows:

> > **NIL** The system will preserve old copyright information, but will not ask you about copyrighting new files. This is the default value of COPYRIGHTFLG.

> > **T** When a file is made, if it has no copyright information, the system will ask you to specify the copyright owner.

> > **NEVER** The system will neither prompt for new copyright information nor preserve old copyright information.

> > **DEFAULT** The value of DEFAULTCOPYRIGHTOWNER (below) is used for putting copyright information in files that don't have any other copyright. The prompt "Copyright owner for file xx:" will still be printed, but the default will be filled in immediately.

COPYRIGHTOWNERS   [Variable]

> COPYRIGHTOWNERS is a list of entries of the form (*KEY OWNERSTRING*), where *KEY* is used as a response to ASKUSER and *OWNERSTRING* is a string which is the full identification of the owner.

DEFAULTCOPYRIGHTOWNER   [Variable]

> If you do not respond in DWIMWAIT seconds to the copyright query, the value of DEFAULTCOPYRIGHTOWNER is used.

Functions Used Within Source Files

The following functions are normally only used within symbolic files, to set variable values, property values, etc. Most of these have special behavior depending on file manager variables.

(**RPAQ** *VAR VALUE*) [NLambda Function]

> An nlambda function like SETQ that sets the top level binding of *VAR* (unevaluated) to *VALUE*.

(**RPAQQ** *VAR VALUE*) [NLambda Function]

> An nlambda function like SETQQ that sets the top level binding of *VAR* (unevaluated) to *VALUE* (unevaluated).

(**RPAQ?** *VAR VALUE*) [NLambda Function]

> Similar to RPAQ, except that it does nothing if *VAR* already has a top level value other than NOBIND. Returns *VALUE* if *VAR* is reset, otherwise NIL.

RPAQ, RPAQQ, and RPAQ? generate errors if X is not a symbol. All are affected by the value of DFNFLG (see Chapter 10). If DFNFLG = ALLPROP (and the value of *VAR* is other than NOBIND), instead of setting X, the corresponding value is stored on the property list of *VAR* under the property VALUE. All are undoable.

(**ADDTOVAR** *VAR* $X_1 X_2 \ldots X_N$) [NLambda NoSpread Function]

> Each X_i that is not a member of the value of *VAR* is added to it, i.e. after ADDTOVAR completes, the value of *VAR* will be (UNION (LIST $X_1 X_2 \ldots X_N$) *VAR*). ADDTOVAR is used by PRETTYDEF for implementing the ADDVARS command. It performs some file manager related operations, i.e. "notices" that *VAR* has been changed. Returns the atom *VAR* (not the value of *VAR*).

(**APPENDTOVAR** *VAR* $X_1 X_2 \ldots X_N$) [NLambda NoSpread Function]

> Similar to ADDTOVAR, except that the values are added to the end tof the list, rather than at the beginning.

(**PUTPROPS** *ATM* $PROP_1 VAL_1 \ldots PROP_N VAL_N$) [NLambda NoSpread Function]

> Nlambda nospread version of PUTPROP (none of the arguments are evaluated). For i = $1 \ldots N$, puts property $PROP_i$, value $VALi$, on the property list of *ATM*. Performs some file manager related operations, i.e., "notices" that the corresponding properties have been changed.

(**SAVEPUT** *ATM PROP VAL*) [Function]

> Same as PUTPROP, but marks the corresponding property value as having been changed (used by the file manager).

File Maps

A file map is a data structure which contains a symbolic 'map' of the contents of a file. Currently, this consists of the begin and end byte address (see GETFILEPTR, in Chapter 25) for each DEFINEQ expression in the file, the begin and end address for each function definition within the DEFINEQ, and the begin and end address for each compiled function.

MAKEFILE, PRETTYDEF, LOADFNS, RECOMPILE, and numerous other system functions depend heavily on the file map for efficient operation. For example, the file map enables LOADFNS to load selected function definitions simply by setting the file pointer to the corresponding address using SETFILEPTR, and then performing a single READ. Similarly, the file map is heavily used by the "remake" option of MAKEFILE (see the Remaking a Symbolic File section): those function definitions that have been changed since the previous version are prettyprinted; the rest are simply copied from the old file to the new one, resulting in a considerable speedup.

Whenever a file is written by MAKEFILE, a file map for the new file is built. Building the map in this case essentially comes for free, since it requires only reading the current file pointer before and after each definition is written or copied. However, building the map does require that PRETTYPRINT *know* that it is printing a DEFINEQ expression. For this reason, you should never print a DEFINEQ expression onto a file yourself, but should instead always use the FNS file manager command (see the Functions and Macros section).

The file map is stored on the property list of the root name of the file, under the property FILEMAP. In addition, MAKEFILE writes the file map on the file itself. For cosmetic reasons, the file map is written as the last expression in the file. However, the *address* of the file map in the file is (over)written into the FILECREATED expression that appears at the beginning of the file so that the file map can be rapidly accessed without having to scan the entire file. In most cases, LOAD and LOADFNS do not have to build the file map at all, since a file map will usually appear in the corresponding file, unless the file was written with BUILDMAPFLG = NIL, or was written outside of Interlisp.

Currently, file maps for *compiled* files are not written onto the files themselves. However, LOAD and LOADFNS will build maps for a compiled file when it is loaded, and store it on the property FILEMAP. Similary, LOADFNS will obtain and use the file map for a compiled file, when available.

The use and creation of file maps is controlled by the following variables:

BUILDMAPFLG   [Variable]

> Whenever a file is read by LOAD or LOADFNS, or written by MAKEFILE, a file map is automatically built unless BUILDMAPFLG = NIL. (BUILDMAPFLG is initially T.)

> While building the map will not help the first reference to a file, it will help in future references. For example, if you perform (LOADFROM 'FOO) where FOO does not contain a file map, the LOADFROM will be (slightly) slower than if FOO did contain a file map, but subsequent calls to LOADFNS for this version of FOO will be able to use the map that was built as the result of the LOADFROM, since it will be stored on FOO's FILEMAP property.

USEMAPFLG [Variable]

If `USEMAPFLG` = `T` (the initial setting), the functions that use file maps will first check the `FILEMAP` property to see if a file map for this file was previously obtained or built. If not, the first expression on the file is checked to see if it is a `FILECREATED` expression that also contains the address of a file map. If the file map is not on the `FILEMAP` property or in the file, a file map will be built (unless `BUILDMAPFLG` = `NIL`).

If `USEMAPFLG` = `NIL`, the `FILEMAP` property and the file will not be checked for the file map. This allows you to recover in those cases where the file and its map for some reason do not agree. For example, if you use a text editor to change a symbolic file that contains a map (not recommended), inserting or deleting just one character will throw that map off. The functions which use file maps contain various integrity checks to enable them to detect that something is wrong, and to generate the error `FILEMAP DOES NOT AGREE WITH CONTENTS OF` *FILE*. In such cases, you can set `USEMAPFLG` to `NIL`, causing the map contained in the file to be ignored, and then reexecute the operation.

18. COMPILER

The compiler is contained in the standard Medley system. It may be used to compile functions defined in Medley, or to compile definitions stored in a file. The resulting compiled code may be stored as it is compiled, so as to be available for immediate use, or it may be written onto a file for subsequent loading.

The most common way to use the compiler is to use one of the file package functions, such as MAKEFILE (Chapter 17), which automatically updates source files, and produces compiled versions. However, it is also possible to compile individual functions defined in Medley, by directly calling the compiler using functions such as COMPILE. No matter how the compiler is called, the function COMPSET is called which asks you certain questions concerning the compilation. (COMPSET sets the free variables LAPFLG, STRF, SVFLG, LCFIL and LSTFIL which determine various modes of operation.) Those that can be answered "yes" or "no" can be answered with YES, Y, or T for "yes"; and NO, N, or NIL for "no". The questions are:

LISTING? This asks whether to generate a listing of the compiled code. The LAP and machine code are usually not of interest but can be helpful in debugging macros. Possible answers are:

   1 Prints output of pass 1, the LAP macro code

   2 Prints output of pass 2, the machine code

  YES Prints output of both passes

  NO Prints no listings

 The variable LAPFLG is set to the answer.

FILE: This question (which only appears if the answer to LISTING? is affirmative) ask where the compiled code listing(s) should be written. Answering T will print the listings at the terminal. The variable LSTFIL is set to the answer.

REDEFINE? This question asks whether the functions compiled should be redefined to their compiled definitions. If this is answered YES, the compiled code is stored and the function definition changed, otherwise the function definition remains unchanged.

 The compiler does *not* respect the value of DFNFLG (Chapter 10) when it redefines functions to their compiled definitions. Therefore, if you set DFNFLG to PROP to completely avoid inadvertantly redefining something in your running system, you *must* not answer YES to this question.

 The variable STRF is set to T (if this is answered YES) or NIL.

SAVE EXPRS?   This question asks whether the original defining EXPRs of functions should be saved.   If answered YES, then before redefining a function to its compiled definition, the EXPR definition is saved on the property list of the function name. Otherwise they are discarded.

It is very useful to save the EXPR definitions, just in case the compiled function needs to be changed. The editing functions will retrieve this saved definition if it exists, rather than reading from a source file.

The variable SVFLG is set to T (if this is answered YES) or NIL.

OUTPUT FILE?   This question asks whether (and where) the compiled definitions should be written into a file for later loading. If you answer with the name of a file, that file will be used. If you answer Y or YES, you will be asked the name of the file. If the file named is already open, it will continue to be used. If you answer T or TTY:, the output will be typed on the teletype (not particularly useful). If you answer N, NO, or NIL, output will *not* be done.

The variable LCFIL is set to the name of the file.

To make answering these questions easier, there are four other possible answers to the LISTING? question, which specify common compiling modes:

S   Same as last setting.   Uses the same answers to compiler questions as given for the last compilation.

F   Compile to File, without redefining functions.

ST   STore new definitions, saving EXPR definitions.

STF   STore new definitions; Forget EXPR definitions.

Implicit in these answers are the answers to the questions on disposition of compiled code and EXPR definitions, so the questions REDEFINE? and SAVE EXPRS? would not be asked if these answers were given. OUTPUT FILE? would still be asked, however. For example:

```
←COMPILE((FACT FACT1 FACT2))
LISTING? ST
OUTPUT FILE? FACT.DCOM
(FACT COMPILING)
.
.
(FACT REDEFINED)
.
.
(FACT2 REDEFINED)
(FACT FACT1 FACT2)
←
```

This process caused the functions `FACT`, `FACT1`, and `FACT2` to be compiled, redefined, and the compiled definitions also written on the file `FACT.DCOM` for subsequent loading.

Compiler Printout

In Medley, for each function *FN* compiled, whether by `TCOMPL`, `RECOMPILE`, or `COMPILE`, the compiler prints:

$$(FN\ (ARG_1 \ldots ARG_N)\ (\text{uses}: VAR_1 \ldots VAR_N)\ (\text{calls}:\ FN_1 \ldots FN_N))$$

The message is printed at the beginning of the second pass of the compilation of *FN*. (ARG_1 **...** ARG_N) is the list of arguments to *FN*; following uses: are the free variables referenced or set in *FN* (not including global variables); following calls: are the undefined functions called within *FN*.

If the compilation of *FN* causes the generation of one or more auxilary functions, a compiler message will be printed for these functions before the message for *FN*, e.g.,

```
(FOOA0027 (X) (uses: XX))
(FOO (A B))
```

When compiling a block, the compiler first prints ($BLKNAME\ BLKFN_1\ BLKFN_2$ ...). Then the normal message is printed for the entire block. The names of the arguments to the block are generated by suffixing # and a number to the block name, e.g., (`FOOBLOCK` (`FOOBLOCK#0 FOOBLOCK#1`) *FREE-VARIABLES*). Then a message is printed for each *entry* to the block.

In addition to the above output, both `RECOMPILE` and `BRECOMPILE` print the name of each function that is being copied from the old compiled file to the new compiled file. The normal compiler message is printed for each function that is actually compiled.

The compiler prints out error messages when it encounters problems compiling a function. For example:

```
----- In BAZ:
***** (BAZ - illegal RETURN)
-----
```

The above error message indicates that an `illegal RETURN` compiler error occurred while trying to compile the function `BAZ`. Some compiler errors cause the compilation to terminate, producing nothing; however, there are other compiler errors which do not stop compilation. The compiler error messages are described in the last section of this chapter.

Compiler printout and error messages go to the file `COUTFILE`, initially `T`. `COUTFILE` can also be set to the name of a file opened for output, in which case all compiler printout will go to `COUTFILE`, i.e.

the compiler will compile "silently." However, any error messages will be printed to both `COUTFILE` as well as `T`.

Global Variables

Variables that appear on the list `GLOBALVARS`, or have the property `GLOBALVAR` with value `T`, or are declared with the `GLOBALVARS` file package command, are called global variables. Such variables are always accessed through their top level value when they are used freely in a compiled function. In other words, a reference to the value of a global variable is equivalent to calling `GETTOPVAL` on the variable, regardless of whether or not it is bound in the current access chain. Similarly, `(SETQ VARIABLE VALUE)` will compile as `(SETTOPVAL (QUOTE VARIABLE) VALUE)`.

All system parameters, unless otherwise specified, are declared as global variables. Thus, *rebinding* these variables in a deep bound system like Medley will not affect the behavior of the system: instead, the variables must be *reset* to their new values, and if they are to be restored to their original values, reset again. For example, you might write

        (SETQ GLOBALVARIABLE NEWVALUE)
        FORM
        (SETQ GLOBALVARIABLE OLDVALUE)

In this case, if an error occurred during the evaluation of *FORM*, or a Control-D was typed, the global variable would not be restored to its original value. The function `RESETVAR` provides a convenient way of resetting global variables in such a way that their values are restored even if an error occurred or Control-D is typed.

> Note: The variables that a given function accesses as global variables can be determined by using the function `CALLS`.

Local Variables and Special Variables

In normal compiled and interpreted code, all variable bindings are accessible by lower level functions because the variable's name is associated with its value. We call such variables *special* variables, or specvars. As mentioned earlier, the block compiler normally does *not* associate names with variable values. Such unnamed variables are not accessible from outside the function which binds them and are therefore *local* to that function. We call such unnamed variables local variables, or localvars.

The time economies of local variables can be achieved without block compiling by use of declarations. Using local variables will increase the speed of compiled code; the price is the work of writing the necessary specvar declarations for those variables which need to be accessed from outside the block.

`LOCALVARS` and `SPECVARS` are variables that affect compilation. During regular compilation, `SPECVARS` is normally `T`, and `LOCALVARS` is `NIL` or a list. This configuration causes all variables

bound in the functions being compiled to be treated as special *except* those that appear on LOCALVARS. During block compilation, LOCALVARS is normally T and SPECVARS is NIL or a list. All variables are then treated as local *except* those that appear on SPECVARS.

Declarations to set LOCALVARS and SPECVARS to other values, and therefore affect how variables are treated, may be used at several levels in the compilation process with varying scope.

1. The declarations may be included in the filecoms of a file, by using the LOCALVARS and SPECVARS file package commands. The scope of the declaration is then the entire file:

    ```
    ... (LOCALVARS . T) (SPECVARS X Y) ...
    ```

2. The declarations may be included in block declarations; the scope is then the block, e.g.,

    ```
    (BLOCKS ((FOOBLOCK FOO FIE (SPECVARS . T) (LOCALVARS
    X)))
    ```

3. The declarations may also appear in individual functions, or in PROG's or LAMBDA's within a function, using the DECLARE function. In this case, the scope of the declaration is the function or the PROG or LAMBDA in which it appears. LOCALVARS and SPECVARS declarations must appear immediately after the variable list in the function, PROG, or LAMBDA, but intervening comments are permitted. For example:

    ```
    (DEFINEQ ((FOO
        (LAMBDA (X Y)
            (DECLARE (LOCALVARS Y))
            (PROG (X Y Z)
                (DECLARE (LOCALVARS X))
                ... ]
    ```

If the above function is compiled (non-block), the outer X will be special, the X bound in the PROG will be local, and both bindings of Y will be local.

Declarations for LOCALVARS and SPECVARS can be used in two ways: either to cause variables to be treated the same whether the function(s) are block compiled or compiled normally, or to affect one compilation mode while not affecting the default in the other mode. For example:

```
(LAMBDA (X Y)
    (DECLARE (SPECVARS . T))
    (PROG (Z) ... ]
```

will cause X, Y, and Z to be specvars for both block and normal compilation while

```
(LAMBDA (X Y)
    (DECLARE (SPECVARS X))
    ... ]
```

will make X a specvar when block compiling, but when regular compiling the declaration will have no effect, because the default value of specvars would be T, and therefore *both* X and Y will be specvars by default.

Although LOCALVARS and SPECVARS declarations have the same form as other components of block declarations such as (LINKFNS . T), their operation is somewhat different because the two variables are not independent. (SPECVARS . T) will cause SPECVARS to be set to T, and LOCALVARS to be set to NIL. (SPECVARS V1 V2 ...) will have *no* effect if the value of SPECVARS is T, but if it is a list (or NIL), SPECVARS will be set to the union of its prior value and (V1 V2 ...). The operation of LOCALVARS is analogous. Thus, to affect both modes of compilation one of the two (LOCALVARS or SPECVARS) must be declared T before specifying a list for the other.

> Note: The variables that a given function binds as local variables or accesses as special variables can be determined by using the function CALLS.

> Note: LOCALVARS and SPECVARS declarations affect the compilation of local variables within a function, but the arguments to functions are always accessible as specvars. This can be changed by redefining the following function:

(**DASSEM.SAVELOCALVARS** *FN*)   [Function]

> This function is called by the compiler to determine whether argument information for *FN* should be written on the compiled file for *FN*. If it returns NIL, the argument information is *not* saved, and the function is stored with arguments U, V, W, etc instead of the originals.

> Initially, DASSEM.SAVELOCALVARS is defined to return T. (MOVD 'NILL 'DASSEM.SAVELOCALVARS) causes the compiler to retain no local variable or argument names. Alternatively, DASSEM.SAVELOCALVARS could be redefined as a more complex predicate, to allow finer discrimination.

Constants

Interlisp allows the expression of constructions which are intended to be description of their constant values. The following functions are used to define constant values. The function SELECTC provides a mechanism for comparing a value to a number of constants.

(**CONSTANT** *X*)   [Function]

> This function enables you to define that the expression *X* should be treated as a "constant" value. When CONSTANT is interpreted, *X* is evaluted each time it is encountered. If the CONSTANT form is compiled, however, the expression will be evaluated only once.

> If the value of *X* has a readable print name, then it will be evaluated at compile-time, and the value will be saved as a literal in the compiled function's definition, as if (QUOTE *VALUE-OF-EXPRESSION*) had appeared instead of (CONSTANT *EXPRESSION*).

> If the value of *X* does not have a readable print name, then the expression *X* itself will be saved with the function, and it will be evaluated when the function is first loaded. The

value will then be stored in the function's literals, and will be retrieved on future references.

If a program needed a list of 30 NILs, you could specify `(CONSTANT (to 30 collect NIL))` instead of `(QUOTE (NIL NIL ...))`. The former is more concise and displays the important parameter much more directly than the latter.

CONSTANT can also be used to denote values that cannot be quoted directly, such as `(CONSTANT (PACK NIL))`, `(CONSTANT (ARRAY 10))`. It is also useful to parameterize quantities that are constant at run time but may differ at compile time, e.g., `(CONSTANT BITSPERWORD)` in a program is exactly equivalent to 36, if the variable BITSPERWORD is bound to 36 when the CONSTANT expression is evaluated at compile time.

Whereas the function CONSTANT attempts to evaluate the expression as soon as possible (compile-time, load-time, or first-run-time), other options are available, using the folowing two function:

(**LOADTIMECONSTANT** X) [Function]

Similar to CONSTANT, except that the evaluation of X is deferred until the compiled code for the containing function is loaded in. For example, `(LOADTIMECONSTANT (DATE))` will return the date the code was loaded. If LOADTIMECONSTANT is interpreted, it merely returns the value of X.

(**DEFERREDCONSTANT** X) [Function]

Similar to CONSTANT, except that the evaluation of X is always deferred until the compiled function is first run. This is useful when the storage for the constant is excessive so that it shouldn't be allocated until (unless) the function is actually invoked. If DEFERREDCONSTANT is interpreted, it merely returns the value of X.

(**CONSTANTS** VAR_1 VAR_2 ... VAR_N) [NLambda NoSpread Function]

Defines VAR_1, ... VAR_N (unevaluated) to be compile-time constants. Whenever the compiler encounters a (free) reference to one of these constants, it will compile the form `(CONSTANT `VAR_i`)` instead.

If VAR_i is a list of the form `(VAR FORM)`, a free reference to the variable will compile as `(CONSTANT `$FORM$`)`.

The compiler prints a warning if user code attempts to bind a variable previously declared as a constant.

Constants can be saved using the CONSTANTS file package command.

Compiling Function Calls

When compiling the call to a function, the compiler must know the type of the function, to determine how the arguments should be prepared (evaluated/unevaluated, spread/nospread). There are three seperate cases: lambda, nlambda spread, and nlambda nospread functions.

To determine which of these three cases is appropriate, the compiler will first look for a definition among the functions in the file that is being compiled. The function can be defined anywhere in any of the files given as arguments to BCOMPL, TCOMPL, BRECOMPILE or RECOMPILE. If the function is not contained in the file, the compiler will look for other information in the variables NLAMA, NLAML, and LAMS, which can be set by you:

NLAMA   [Variable]

(For NLAMbda Atoms) A list of functions to be treated as nlambda nospread functions by the compiler.

NLAML   [Variable]

(For NLAMbda List) A list of functions to be treated as nlambda spread functions by the compiler.

LAMS   [Variable]

A list of functions to be treated as lambda functions by the compiler. Note that including functions on LAMS is only necessary to override in-core nlambda definitions, since in the absence of other information, the compiler assumes the function is a lambda.

If the function is not contained in a file, or on the lists NLAMA, NLAML, or LAMS, the compiler will look for a current definition in the Interlisp system, and use its type. If there is no current definition, next COMPILEUSERFN is called:

COMPILEUSERFN   [Variable]

When compiling a function call, if the function type cannot be found by looking in files, the variables NLAMA, NLAML, or LAMS, or at a current definition, then if the value of COMPILEUSERFN is not NIL, the compiler calls (the value of) COMPILEUSERFN giving it as arguments CDR of the form and the form itself, i.e., the compiler does (APPLY* COMPILEUSERFN (CDR FORM) FORM). If a non-NIL value is returned, it is compiled instead of FORM. If NIL is returned, the compiler compiles the original expression as a call to a lambda spread that is not yet defined.

COMPILEUSERFN is only called when the compiler encounters a *list* CAR of which is not the name of a defined function. You can instruct the compiler about how to compile other data types via COMPILETYPELST.

CLISP uses COMPILEUSERFN to tell the compiler how to compile iterative statements, IF-THEN-ELSE statements, and pattern match constructs.

If the compiler cannot determine the function type by any of the means above, it assumes that the function is a lambda function, and its arguments are to be evaluated.

If there are nlambda functions called from the functions being compiled, and they are only defined in a separate file, they must be included on NLAMA or NLAML, or the compiler will incorrectly assume that their arguments are to be evaluated, and compile the calling function correspondingly. This is only necessary if the compiler does not "know" about the function. If the function is defined at compile time, or is handled via a macro, or is contained in the same group of files as the functions that call it, the compiler will automatically handle calls to that function correctly.

FUNCTION and Functional Arguments

Compiling the function FUNCTION may involve creating and compiling a seperate "auxiliary function", which will be called at run time. An auxiliary function is named by attaching a GENSYM to the end of the name of the function in which they appear, e.g., FOOA0003. For example, suppose FOO is defined as (LAMBDA (X) ... (FOO1 X (FUNCTION ...)) ...) and compiled. When FOO is run, FOO1 will be called with two arguments, X, and FOOA000*N* and FOO1 will call FOOA000*N* each time it uses its functional argument.

Compiling FUNCTION will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (MAPCAR, MAPLIST, etc.). A considerable savings in time could be achieved by making FOO1 compile open via a computed macro, e.g.

```
(PUTPROP 'FOO1 'MACRO
      '(Z (LIST (SUBST (CADADR Z)
                   (QUOTE FN)
                   DEF)
            (CAR Z)))
```

DEF is the definition of FOO1 as a function of just its first argument, and FN is the name used for its functional argument in its definition. In this case, (FOO1 X (FUNCTION ...)) would compile as an expression, containing the argument to FUNCTION as an open LAMBDA expression. Thus you save not only the function call to FOO1, but also each of the function calls to its functional argument. For example, if FOO1 operates on a list of length ten, eleven function calls will be saved. Of course, this savings in time costs space, and you must decide which is more important.

Open Functions

When a function is called from a compiled function, a system routine is invoked that sets up the parameter and control push lists as necessary for variable bindings and return information. If the amount of time spent *inside* the function is small, this function calling time will be a significant percentage of the total time required to use the function. Therefore, many "small" functions, e.g., CAR, CDR, EQ, NOT, CONS are always compiled "open", i.e., they do not result in a function call. Other larger

functions such as PROG, SELECTQ, MAPC, etc. are compiled open because they are frequently used. You can make other functions compile open via MACRO definitions. You can also affect the compiled code via COMPILEUSERFN and COMPILETYPELST.

COMPILETYPELST

Most of the compiler's mechanism deals with how to handle forms (lists) and variables (symbols). You can affect the compiler's behaviour with respect to lists and literal atoms in a number of ways, e.g. macros, declarations, COMPILEUSERFN, etc. COMPILETYPELST allows you to tell the compiler what to do when it encounters a data type *other* than a list or an atom. It is the facility in the compiler that corresponds to DEFEVAL for the interpreter.

> **COMPILETYPELST**   [Variable]
>
>> A list of elements of the form (*TYPENAME* . *FUNCTION*). Whenever the compiler encounters a datum that is not a list and not an atom (or a number) in a context where the datum is being evaluated, the type name of the datum is looked up on COMPILETYPELST. If an entry appears CAR of which is equal to the type name, CDR of that entry is applied to the datum. If the value returned by this application is *not* EQ to the datum, then that value is compiled instead. If the value *is* EQ to the datum, or if there is no entry on COMPILETYPELST for this type name, the compiler simply compiles the datum as (QUOTE *DATUM*).

Compiling CLISP

Since the compiler does not know about CLISP, in order to compile functions containing CLISP constructs, the definitions must first be DWIMIFYed. You can automate this process in several ways:

1.   If the variable DWIMIFYCOMPFLG is T, the compiler will always DWIMIFY expressions before compiling them. DWIMIFYCOMPFLG is initially NIL.

2.   If a file has the property FILETYPE with value CLISP on its property list, TCOMPL, BCOMPL, RECOMPILE, and BRECOMPILE will operate as though DWIMIFYCOMPFLG is T and DWIMIFY all expressions before compiling.

3.   If the function definition has a local CLISP declaration, including a null declaration, i.e., just (CLISP:), the definition will be automatically DWIMIFYed before compiling.

> Note: COMPILEUSERFN is defined to call DWIMIFY on iterative statements, IF-THEN statements, and fetch, replace, and match expressions, i.e., any CLISP construct which can be recognized by its CAR of form. Thus, if the only CLISP constructs in a function appear inside of iterative statements, IF statements, etc., the function does not have to be dwimified before compiling.

If DWIMIFY is ever unsuccessful in processing a CLISP expression, it will print the error message UNABLE TO DWIMIFY followed by the expression, and go into a break unless DWIMESSGAG = T. In this case, the expression is just compiled as is, i.e. as though CLISP had not been enabled. You can exit the break in one of these ways:

1. Type OK to the break, which will cause the compiler to try again, e.g. you could define some missing records while in the break, and then continue

2. Type ↑, which will cause the compiler to simply compile the expression as is, i.e. as though CLISP had not been enabled in the first place

3. Return an expression to be compiled in its place by using the RETURN break command.

> Note: TCOMPL, BCOMPL, RECOMPILE, and BRECOMPILE all scan the entire file before doing any compiling, and take note of the names of all functions that are defined in the file as well as the names of all variables that are set by adding them to NOFIXFNSLST and NOFIXVARSLST, respectively. Thus, if a function is not currently defined, but *is* defined in the file being compiled, when DWIMIFY is called before compiling, it will not attempt to interpret the function name as CLISP when it appears as CAR of a form. DWIMIFY also takes into account variables that have been declared to be LOCALVARS, or SPECVARS, either via block declarations or DECLARE expressions in the function being compiled, and does not attempt spelling correction on these variables. The declaration USEDFREE may also be used to declare variables simply used freely in a function. These variables will also be left alone by DWIMIFY. Finally, NOSPELLFLG is reset to T when compiling functions from a file (as opposed to from their in-core definition) so as to suppress spelling correction.

Compiler Functions

Normally, the compiler is envoked through file package commands that keep track of the state of functions, and manage a set of files, such as MAKEFILE. However, it is also possible to explicitly call the compiler using one of a number of functions. Functions may be compiled from in-core definitions (via COMPILE), or from definitions in files (TCOMPL), or from a combination of in-core and file definitions (RECOMPILE).

TCOMPL and RECOMPILE produce "compiled" files. Compiled files usually have the same name as the symbolic file they were made from, suffixed with DCOM (the compiled file extension is stored as the value of the variable COMPILE.EXT). The file name is constructed from the name field only, e.g., (TCOMPL '<BOBROW>FOO.TEM;3) produces FOO.DCOM on the connected directory. The version number will be the standard default.

A "compiled file" contains the same expressions as the original symbolic file, except for the following:

1. A special FILECREATED expression appears at the front of the file which contains information used by the file package, and which causes the message COMPILED ON *DATE* to be printed when the file is loaded (the actual string printed is the value of COMPILEHEADER).

2. Every DEFINEQ in the symbolic file is replaced by the corresponding compiled definitions in the compiled file.

3. Expressions following a DONTCOPY tag inside of a DECLARE: that appears in the symbolic file are not copied to the compiled file.

The compiled definitions appear at the front of the compiled file, i.e., before the other expressions in the symbolic file, *regardless of where they appear in the symbolic file.* The only exceptions are expressions that follow a FIRST tag inside of a DECLARE:. This "compiled" file can be loaded into any Interlisp system with LOAD.

> Note: When a function is compiled from its in-core definition (as opposed to being compiled from a definition in a file), and the function has been modified by BREAK, TRACE, BREAKIN, or ADVISE, it is first restored to its original state, and a message is printed out, e.g., FOO UNBROKEN. If the function is not defined by an expr definition, the value of the function's EXPR property is used for the compilation, if there is one. If there is no EXPR property, and the compilation is being performed by RECOMPILE, the definition of the function is obtained from the file (using LOADFNS). Otherwise, the compiler prints (*FN* NOT COMPILEABLE), and goes on to the next function.

(**COMPILE** *X FLG*)   [Function]

> *X* is a list of functions (if atomic, (LIST *X*) is used). COMPILE first asks the standard compiler questions, and then compiles each function on *X*, using its in-core definition. Returns *X*.
>
> If compiled definitions are being written to a file, the file is closed unless *FLG* = T.

(**COMPILE1** *FN DEF*)   [Function]

> Compiles *DEF*, redefining *FN* if STRF = T (STRF is one of the variables set by COMPSET). COMPILE1 is used by COMPILE, TCOMPL, and RECOMPILE.
>
> If DWIMIFYCOMPFLG is T, or *DEF* contains a CLISP declaration, *DEF* is dwimified before compiling.

(**TCOMPL** *FILES*)   [Function]

> TCOMPL is used to "compile files"; given a symbolic LOAD file (e.g., one created by MAKEFILE), it produces a "compiled file". *FILES* is a list of symbolic files to be compiled (if atomic, (LIST *FILES*) is used). TCOMPL asks the standard compiler questions, except for "OUTPUT FILE:". The output from the compilation of each symbolic file is written on a file of the same name suffixed with DCOM, e.g., (TCOMPL '(SYM1 SYM2)) produces two files, SYM1.DCOM and SYM2.DCOM.

TCOMPL processes the files one at a time, reading in the entire file. For each FILECREATED expression, the list of functions that were marked as changed by the file package is noted, and the FILECREATED expression is written onto the output file. For each DEFINEQ expression, TCOMPL adds any nlambda functions defined in the DEFINEQ to NLAMA or NLAML, and adds lambda functions to LAMS, so that calls to these functions will be compiled correctly. NLAMA, NLAML, and LAMS are rebound to their top level values (using RESETVAR) by all of the compiling functions, so that any additions to these lists while inside of these functions will not propagate outside. Expressions beginning with DECLARE: are processed specially. All other expressions are collected to be subsequently written onto the output file.

After processing the file in this fashion, TCOMPL compiles each function, except for those functions which appear on the list DONTCOMPILEFNS (initially NIL), and writes the compiled definition onto the output file. TCOMPL then writes onto the output file the other expressions found in the symbolic file. DONTCOMPILEFNS might be used for functions that compile open, since their definitions would be superfluous when operating with the compiled file. Note that DONTCOMPILEFNS can be set via block declarations.

> Note: If the rootname of a file has the property FILETYPE with value CLISP, or value a list containing CLISP, TCOMPL rebinds DWIMIFYCOMPFLG to T while compiling the functions on *FILE*, so the compiler will DWIMIFY all expressions before compiling them.

TCOMPL returns a list of the names of the output files. All files are properly terminated and closed. If the compilation of any file is aborted via an error or Control-D, all files are properly closed, and the (partially complete) compiled file is deleted.

(**RECOMPILE** *PFILE CFILE FNS*)   [Function]

The purpose of RECOMPILE is to allow you to update a compiled file without recompiling every function in the file. RECOMPILE does this by using the results of a previous compilation. It produces a compiled file similar to one that would have been produced by TCOMPL, but at a considerable savings in time by only compiling selected functions, and copying the compiled definitions for the remainder of the functions in the file from an earlier TCOMPL or RECOMPILE file.

PFILE is the name of the **P**retty file (source file) to be compiled; *CFILE* is the name of the **C**ompiled file containing compiled definitions that may be copied. *FNS* indicates which functions in *PFILE* are to be recompiled, e.g., have been changed or defined for the first time since *CFILE* was made. Note that *PFILE*, not *FNS*, drives RECOMPILE.

RECOMPILE asks the standard compiler questions, except for "OUTPUT FILE:". As with TCOMPL, the output automatically goes to *PFILE*.DCOM. RECOMPILE processes *PFILE* the same as does TCOMPL except that DEFINEQ expressions are not actually read into core. Instead, RECOMPILE uses the filemap to obtain a list of the functions contained in *PFILE*. The filemap enables RECOMPILE to skip over the DEFINEQs in the file by simply resetting the file pointer, so that in most cases the scan of the symbolic file is very fast (the only processing required is the reading of the non-DEFINEQs and the processing of the DECLARE: expressions as with TCOMPL). A map is built if the symbolic file does not

already contain one, for example if it was written in an earlier system, or with `BUILDMAPFLG = NIL`.

After this initial scan of `PFILE`, `RECOMPILE` then processes the functions defined in the file. For each function in `PFILE`, `RECOMPILE` determines whether or not the function is to be (re)compiled. Functions that are members of `DONTCOMPILEFNS` are simply ignored. Otherwise, a function is recompiled if :

1. *FNS* is a list and the function is a member of that list

2. *FNS* = `T` or `EXPRS` and the function is defined by an expr definition

3. *FNS* = `CHANGES` and the function is marked as having been changed in the `FILECREATED` expression in *PFILE*

4. *FNS* = `ALL`

If a function is not to be recompiled, `RECOMPILE` obtains its compiled definition from *CFILE*, and copies it (and all generated subfunctions) to the output file, `PFILE`.`DCOM`. If the function does not appear on *CFILE*, `RECOMPILE` simply recompiles it. Finally, after processing all functions, `RECOMPILE` writes out all other expressions that were collected in the prescan of *PFILE*.

> Note: If *FNS* = `ALL`, *CFILE* is superfluous, and does not have to be specified. This option may be used to compile a symbolic file that has never been compiled before, but which has already been loaded (since using `TCOMPL` would require reading the file in a second time).

If *CFILE* = `NIL`, `PFILE`.`DCOM` (the old version of the output file) is used for copying *from*. If both *FNS* and *CFILE* are `NIL`, *FNS* is set to the value of `RECOMPILEDEFAULT`, which is initially `CHANGES`. Thus you can perform his edits, dump the file, and then simply (`RECOMPILE 'FILE`) to update the compiled file.

The value of `RECOMPILE` is the file name of the new compiled file, `PFILE`.`DCOM`. If `RECOMPILE` is aborted due to an error or Control-D, the new (partially complete) compiled file will be closed and deleted.

`RECOMPILE` is designed to allow you to conveniently and *efficiently* update a compiled file, even when the corresponding symbolic file has not been (completely) loaded. For example, you can perform a `LOADFROM` to "notice" a symbolic file, edit the functions he wants to change (the editor will automatically load those functions not already loaded), call `MAKEFILE` to update the symbolic file (`MAKEFILE` will copy the unchanged functions from the old symbolic file), and then perform (`RECOMPILE PFILE`).

> Note: Since `PRETTYDEF` automatically outputs a suitable `DECLARE`: expression to indicate which functions in the file (if any) are defined as `NLAMBDA`s, calls to these functions will be handled correctly, even though the `NLAMBDA` functions themselves may never be loaded, or even looked at, by `RECOMPILE`.

Block Compiling

In Interlisp-10, block compiling provides a way of compiling several functions into a single block. Function calls between the component functions of the block are very fast. Thus, compiling a block consisting of just a single recursive function may be yield great savings if the function calls itself many times. The output of a block compilation is a single, usually large, function. Calls from within the block to functions outside of the block look like regular function calls. A block can be entered via several different functions, called entries. These must be specified when the block is compiled.

In Medley, block compiling is handled somewhat differently; block compiling provides a mechanism for hiding function names internal to a block, but it does not provide a performance improvement. Block compiling in Medley works by automatically renaming the block functions with special names, and calling these functions with the normal function-calling mechanisms. Specifically, a function *FN* is renamed to `\BLOCK-NAME/FN`. For example, function FOO in block BAR is renamed to `\BAR/FOO`. Note that it is possible with this scheme to break functions internal to a block.

Block Declarations

Block compiling a file frequently involves giving the compiler a lot of information about the nature and structure of the compilation, e.g., block functions, entries, specvars, etc. To help with this, there is the BLOCKS file package command, which has the form:

$$(\text{BLOCKS } BLOCK_1 ... BLOCK_N)$$

where each $BLOCK_i$ is a block declaration. The BLOCKS command outputs a DECLARE: expression, which is noticed by BCOMPL and BRECOMPILE. BCOMPL and BRECOMPILE are sensitive to these declarations and take the appropriate action.

> Note: Masterscope includes a facility for checking the block declarations of a file or files for various anomalous conditions, e.g. functions in block declarations which aren't on the file(s), functions in ENTRIES not in the block, variables that may not need to be SPECVARS because they are not used freely below the places they are bound, etc.

A block declaration is a list of the form:

$$(BLKNAME \ BLKFN_1 ... BLKFN_M$$
$$(VAR_1 \ . \ VALUE_1) ... (VAR_N \ . \ VALUE_N))$$

$BLKNAME$ is the name of a block. $BLKFN_1 ... BLKFN_M$ are the functions in the block and correspond to $BLKFNS$ in the call to BLOCKCOMPILE. The $(VAR_i \ . \ VALUE_i)$ expressions indicate the settings for variables affecting the compilation of that block. If $VALUE_i$ is atomic, then VAR_i is set to $VALUE_i$, otherwise VAR_i is set to the UNION of $VALUE_i$ and the current value of the variable VAR_i. Also, expressions of the form $(VAR \ * \ FORM)$ will cause $FORM$ to be evaluated and the resulting list used as described above (e.g. $(\text{GLOBALVARS} \ * \ \text{MYGLOBALVARS})$).

For example, consider the block declaration below. The block name is `EDITBLOCK`, it includes a number of functions (`EDITL0,   EDITL1,   ...   EDITH`), and it sets the variables `ENTRIES`, `SPECVARS`, `RETFNS`, and `GLOBALVARS`.

```
(EDITBLOCK
        EDITL0 EDITL1 UNDOEDITL EDITCOM EDITCOMA
        EDITMAC EDITCOMS EDIT]UNDO UNDOEDITCOM EDITH
        (ENTRIES EDITL0 ## UNDOEDITL)
        (SPECVARS L COM LCFLG #1 #2 #3 LISPXBUFS)
        (RETFNS EDITL0)
        (GLOBALVARS EDITCOMSA EDITCOMSL EDITOPS))
```

Whenever `BCOMPL` or `BRECOMPILE` encounter a block declaration, they rebind `RETFNS`, `SPECVARS`, `GLOBALVARS`, `BLKLIBRARY`, and `DONTCOMPILEFNS` to their top level values, bind `BLKAPPLYFNS` and `ENTRIES` to `NIL`, and bind `BLKNAME` to the first element of the declaration. They then scan the rest of the declaration, setting these variables as described above. When the declaration is exhausted, the block compiler is called and given `BLKNAME`, the list of block functions, and `ENTRIES`.

If a function appears in a block declaration, but is not defined in one of the files, then if it has an in-core definition, this definition is used and a message printed `NOT ON FILE,` `COMPILING IN CORE DEFINITION`. Otherwise, the message `NOT  COMPILEABLE,` is printed and the block declaration processed as though the function were not on it, i.e. calls to the function will be compiled as external function calls.

Since all compiler variables are rebound for each block declaration, the declaration only has to set those variables it wants *changed*. Furthermore, setting a variable in one declaration has no effect on the variable's value for another declaration.

After finishing all blocks, `BCOMPL` and `BRECOMPILE` treat any functions in the file that did not appear in a block declaration in the same way as do `TCOMPL` and `RECOMPILE`. If you wish a function compiled separately as well as in a block, or if you wish to compile some functions (not blockcompile), with some compiler variables changed, you can use a special pseudo-block declaration of the form

$$(NIL\ BLKFN_1\ ...\ BLKFN_M\ (VAR_1\ .\ VALUE_1)\ ...\ (VAR_N\ .\ VALUE_N))$$

which means that $BLKFN_1\ ...\ BLKFN_M$ should be compiled after first setting $VAR_1\ ...\ VAR_N$ as described above.

The following variables control other aspects of compiling a block:

RETFNS   [Variable]

> Value is a list of internal block functions whose names must appear on the stack, e.g., if the function is to be returned from `RETFROM`, `RETTO`, `RETEVAL`, etc. Usually, internal calls between functions in a block are not put on the stack.

`BLKAPPLYFNS` [Variable]

Value is a list of internal block functions called by other functions in the same block using `BLKAPPLY` or `BLKAPPLY*` for efficiency reasons.

Normally, a call to `APPLY` from inside a block would be the same as a call to any other function outside of the block. If the first argument to `APPLY` turned out to be one of the entries to the block, the block would have to be reentered. `BLKAPPLYFNS` enables a program to compute the name of a function in the block to be called next, without the overhead of leaving the block and reentering it. This is done by including on the list `BLKAPPLYFNS` those functions which will be called in this fashion, and by using `BLKAPPLY` in place of `APPLY`, and `BLKAPPLY*` in place of `APPLY*`. If `BLKAPPLY` or `BLKAPPLY*` is given a function not on `BLKAPPLYFNS`, the effect is the same as a call to `APPLY` or `APPLY*` and no error is generated. Note however, that `BLKAPPLYFNS` must be set at *compile* time, not run time, and furthermore, that all functions on `BLKAPPLYFNS` must be in the block, or an error is generated (at compile time), `NOT ON BLKFNS`.

`BLKAPPLYFNS` [Variable]

Value is a list of functions that are considered to be in the "block library" of functions that should automatically be included in the block if they are called within the block.

Compiling a function open via a macro provides a way of eliminating a function call. For block compiling, the same effect can be achieved by including the function in the block. A further advantage is that the code for this function will appear only once in the block, whereas when a function is compiled open, its code appears at each place where it is called.

The block library feature provides a convenient way of including functions in a block. It is just a convenience since you can always achieve the same effect by specifying the function(s) in question as one of the block functions, provided it has an expr definition at compile time. The block library feature simply eliminates the burden of supplying this definition.

To use the block library feature, place the names of the functions of interest on the list `BLKLIBRARY`, and their expr definitions on the property list of the functions under the property `BLKLIBRARYDEF`. When the block compiler compiles a form, it first checks to see if the function being called is one of the block functions. If not, and the function is on `BLKLIBRARY`, its definition is obtained from the property value of `BLKLIBRARYDEF`, and it is automatically included as part of the block.

Block Compiling Functions

There are three user level functions for block compiling, `BLOCKCOMPILE`, `BCOMPL`, and `BRECOMPILE`, corresponding to `COMPILE`, `TCOMPL`, and `RECOMPILE`. Note that all of the remarks on macros, globalvars, compiler messages, etc., all apply equally for block compiling. Using block declarations, you can intermix in a single file functions compiled normally and block compiled functions.

(**BLOCKCOMPILE** *BLKNAME BLKFNS ENTRIES FLG*) [Function]

> *BLKNAME* is the name of a block, *BLKFNS* is a list of the functions comprising the block, and *ENTRIES* a list of entries to the block.

> Each of the entries must also be on *BLKFNS* or an error is generated, NOT ON BLKFNS. If only one entry is specified, the block name can also be one of the *BLKFNS*, e.g., (BLOCKCOMPILE 'FOO '(FOO FIE FUM) '(FOO)). However, if more than one entry is specified, an error will be generated, CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME.

> If *ENTRIES* is NIL, (LIST *BLKNAME*) is used, e.g., (BLOCKCOMPILE 'COUNT '(COUNT COUNT1))

> If *BLKFNS* is NIL, (LIST *BLKNAME*) is used, e.g., (BLOCKCOMPILE 'EQUAL)

> BLOCKCOMPILE asks the standard compiler questions, and then begins compiling. As with COMPILE, if the compiled code is being written to a file, the file is closed unless *FLG* = T. The value of BLOCKCOMPILE is a list of the entries, or if *ENTRIES* = NIL, the value is *BLKNAME*.

> The output of a call to BLOCKCOMPILE is one function definition for *BLKNAME*, plus definitions for each of the functions on *ENTRIES* if any. These entry functions are very short functions which immediately call *BLKNAME*.

(**BCOMPL** *FILES CFILE*) [Function]

> *FILES* is a list of symbolic files (if atomic, (LIST *FILES*) is used). BCOMPL differs from TCOMPL in that it compiles all of the files at once, instead of one at a time, in order to permit one block to contain functions in several files. (If you have several files to be BCOMPLed *separately*, you must make several calls to BCOMPL.) Output is to *CFILE* if given, otherwise to a file whose name is (CAR *FILES*) suffixed with DCOM. For example, (BCOMPL '(EDIT WEDIT)) produces one file, EDIT.DCOM.

> BCOMPL asks the standard compiler questions, except for "OUTPUT FILE:", then processes each file exactly the same as TCOMPL. BCOMPL next processes the block declarations as described above. Finally, it compiles those functions not mentioned in one of the block declarations, and then writes out all other expressions.

> If *any* of the files have property FILETYPE with value CLISP, or a list containing CLISP, then DWIMIFYCOMPFLG is rebound to T for *all* of the files.

> The value of BCOMPL is the output file (the new compiled file). If the compilation is aborted due to an error or Control-D, all files are closed and the (partially complete) output file is deleted.

> It is permissible to TCOMPL files set up for BCOMPL; the block declarations will simply have no effect. Similarly, you can BCOMPL a file that does not contain any block declarations and the result will be the same as having TCOMPLed it.

(**BRECOMPILE** *FILES CFILE FNS* —) [Function]

> BRECOMPILE plays the same role for BCOMPL that RECOMPILE plays for TCOMPL. Its purpose is to allow you to update a compiled file without requiring an entire BCOMPL.

> *FILES* is a list of symbolic files (if atomic, (LIST *FILES*) is used). *CFILE* is the compiled file produced by BCOMPL or a previous BRECOMPILE that contains compiled definitions that may be copied. The interpretation of *FNS* is the same as with RECOMPILE.

> BRECOMPILE asks the standard compiler questions, except for "OUTPUT FILE:". As with BCOMPL, output automatically goes to *FILE*.DCOM, where *FILE* is the first file in *FILES*.

> BRECOMPILE processes each file the same as RECOMPILE, then processes each block declaration. If *any* of the functions in the block are to be recompiled, the entire block must be (is) recompiled. Otherwise, the block is copied from *CFILE* as with RECOMPILE. For pseudo-block declarations of the form (NIL FN_1 ...), all variable assignments are made, but only those functions indicated by *FNS* are recompiled.

> After completing the block declarations, BRECOMPILE processes all functions that do not appear in a block declaration, recompiling those dictated by *FNS*, and copying the compiled definitions of the remaining from *CFILE*.

> Finally, BRECOMPILE writes onto the output file the "other expressions" collected in the initial scan of *FILES*.

> The value of BRECOMPILE is the output file (the new compiled file). If the compilation is aborted due to an error or Control-D, all files are closed and the (partially complete) output file is deleted.

> If *CFILE* = NIL, the old version of *FILE*.DCOM is used, as with RECOMPILE. In addition, if *FNS* and *CFILE* are both NIL, *FNS* is set to the value of RECOMPILEDEFAULT, initially CHANGES.

Compiler Error Messages

Messages describing errors in the function being compiled are also printed on the terminal. These messages are always preceded by *****. Unless otherwise indicated below, the compilation will continue.

(*FN* NOT ON FILE, COMPILING IN CORE DEFINITION)

> From calls to BCOMPL and BRECOMPILE.

(*FN* NOT COMPILEABLE)

> An EXPR definition for *FN* could not be found. In this case, no code is produced for *FN*, and the compiler proceeds to the next function to be compiled, if any.

(*FN* NOT FOUND)

>   Occurs when RECOMPILE or BRECOMPILE try to copy the compiled definition of *FN* from
>   *CFILE*, and cannot find it. In this case, no code is copied and the compiler proceeds to the
>   next function to be compiled, if any.

(*FN* NOT ON BLKFNS)

>   *FN* was specified as an entry to a block, or else was on BLKAPPLYFNS, but did not appear
>   on the *BLKFNS*. In this case, no code is produced for the entire block and the compiler
>   proceeds to the next function to be compiled, if any.

(*FN* CAN'T BE BOTH AN ENTRY AND THE BLOCK NAME)

>   In this case, no code is produced for the entire block and the compiler proceeds to the next
>   function to be compiled, if any.

(*BLKNAME* – USED BLKAPPLY WHEN NOT APPLICABLE)

>   BLKAPPLY is used in the block *BLKNAME*, but there are no BLKAPPLYFNS or ENTRIES
>   declared for the block.

(*VAR* SHOULD BE A SPECVAR – USED FREELY BY *FN*)

>   While compiling a block, the compiler has already generated code to bind *VAR* as a
>   LOCALVAR, but now discovers that *FN* uses *VAR* freely. *VAR* should be declared a
>   SPECVAR and the block recompiled.

((* --) COMMENT USED FOR VALUE)

>   A comment appears in a context where its value is being used, e.g. (LIST X (* --)
>   Y). The compiled function will run, but the value at the point where the comment was
>   used is undefined.

((*FORM*) – NON-ATOMIC CAR OF FORM)

>   If you intended to treat the value of *FORM* as a function, you should use APPLY* (Chapter
>   10). *FORM* is compiled as if APPLY* had been used.

((SETQ *VAR EXPR* --) BAD SETQ)

>   SETQ of more than two arguments.

(*FN* – USED AS ARG TO NUMBER FN?)

>   The value of a predicate, such as GREATERP or EQ, is used as an argument to a function
>   that expects numbers, such as IPLUS.

(*FN* – NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT)

>   The compiler has assumed *FN* is the name of a function. If you intended to treat the *value*
>   of *FN* as a function, APPLY* (Chapter 10) should be used. This message is printed when
>   *FN* is not defined, and is also a local variable of the function being compiled.

(*FN* – ILLEGAL RETURN)

    RETURN encountered when not in PROG.

(*TG* – ILLEGAL GO)

    GO encountered when not in a PROG.

(*TG* – MULTIPLY DEFINED TAG)

    TG is a PROG label that is defined more than once in a single PROG. The second definition is ignored.

(*TG* – UNDEFINED TAG)

    TG is a PROG label that is referenced but not defined in a PROG.

(*VAR* – NOT A BINDABLE VARIABLE)

    VAR is NIL, T, or else not a literal atom.

(*VAR VAL* -- BAD PROG BINDING)

    Occurs when there is a prog binding of the form (*VAR VAL$_1$ ... VAL$_N$*).

(*TG* – MULTIPLY DEFINED TAG, LAP)

    TG is a label that was encountered twice during the second pass of the compilation. If this error occurs with no indication of a multiply defined tag during pass one, the tag is in a LAP macro.

(*TG* – UNDEFINED TAG, LAP)

    TG is a label that is referenced during the second pass of compilation and is not defined. LAP treats *TG* as though it were a COREVAL, and continues the compilation.

(*TG* – MULTIPLY DEFINED TAG, ASSEMBLE)

    TG is a label that is defined more than once in an assemble form.

(*TG* – UNDEFINED TAG, ASSEMBLE)

    TG is a label that is referenced but not defined in an assemble form.

(*OP* – OPCODE? – ASSEMBLE)

    OP appears as CAR of an assemble statement, and is illegal.

(NO BINARY CODE GENERATED OR LOADED FOR *FN*)

    A previous error condition was sufficiently serious that binary code for *FN* cannot be loaded without causing an error.

19. DWIM

A surprisingly large percentage of the errors made by Interlisp users are of the type that could be corrected by another Lisp programmer without any information about the purpose of the program or expression in question, e.g., misspellings, certain kinds of parentheses errors, etc. To correct these types of errors we have implemented in Medley a DWIM facility, short for Do-What-I-Mean. DWIM is called automatically whenever an error occurs in the evaluation of an Interlisp expression. (Currently, DWIM only operates on unbound atoms and undefined function errors.) DWIM then proceeds to try to correct the mistake using the current context of computation plus information about what you had previously been doing (and what mistakes you had been making) as guides to the remedy of the error. If DWIM is able to make the correction, the computation continues as though no error had occurred. Otherwise, the procedure is the same as though DWIM had not intervened: a break occurs, or an unwind to the last ERRORSET (see Chapter 14). The following protocol illustrates the operation of DWIM.

For example, suppose you define the factorial function (FACT N) as follows:

```
←DEFINEQ((FACT (LAMBDA (N) (COND
((ZEROP N0 1) ((T (ITIMS N (FACCT 9SUB1 N]
(FACT)
        ←
```

Note that the definition of FACT contains several mistakes: ITIMES and FACT have been misspelled; the 0 in N0 was intended to be a right parenthesis, but the Shift key was not pressed; similarly, the 9 in 9SUB1 was intended to be a left parenthesis; and finally, there is an extra left parenthesis in front of the T that begins the final clause in the conditional.

```
←PRETTYPRNT((FACCT]
        =PRETTYPRINT
        =FACT

        (FACT
          [LAMBDA (N)
            (COND
              ((ZEROP N0 1)
                ((T (ITIMS N (FACCT 9SUB1 N])
        (FACT)

    ←
```

After defining FACT, you want to look at its definition using PRETTYPRINT, which you unfortunately misspell. Since there is no function PRETTYPRNT in the system, an undefined function error occurs, and DWIM is called. DWIM invokes its spelling corrector, which searches a list of functions frequently used (by *this* user) for the best possible match. Finding one that is extremely close, DWIM proceeds on the assumption that PRETTYPRNT meant PRETTYPRINT, notifies you of this, and calls PRETTYPRINT.

At this point, `PRETTYPRINT` would normally print `(FACCT NOT PRINTABLE)` and exit, since `FACCT` has no definition. Note that this is *not* an Interlisp error condition, so that `DWIM` would not be called as described above. However, it is obviously not what you *meant*.

This sort of mistake is corrected by having `PRETTYPRINT` itself explicitly invoke the spelling corrector portion of `DWIM` whenever given a function with no `EXPR` definition. Thus, with the aid of DWIM `PRETTYPRINT` is able to determine that you want to see the definition of the function `FACT`, and proceeds accordingly.

```
←FACT(3]
       N0 [IN FACT] -> N ) ?   YES
       [IN FACT] (COND -- ((T --))) ->
                       (COND -- (T --))
       ITIMS [IN FACT] -> ITIMES
       FACCT [IN FACT] -> FACT
       9SUB1 [IN FACT] ->  ( SUB1 ?  YES
       6

←PP FACT
       (FACT
         [LAMBDA (N)
           (COND
             ((ZEROP N)
               1)
             (T (ITIMES N (FACT (SUB1 N])
       FACT


←
```

You now call `FACT`. During its execution, five errors occur, and `DWIM` is called five times. At each point, the error is corrected, a message is printed describing the action taken, and the computation is allowed to continue as if no error had occurred. Following the last correction, `6` is printed, the value of `(FACT 3)`. Finally, you prettyprint the new, now correct, definition of `FACT`.

In this particular example, you were operating in `TRUSTING` mode, which gives `DWIM` carte blanche for most corrections. You can also operate in `CAUTIOUS` mode, in which case `DWIM` will inform you of intended corrections before they are made, and allow you to approve or disapprove of them. If `DWIM` was operating in `CAUTIOUS` mode in the example above, it would proceed as follows:

```
←FACT(3)
       N0 [IN FACT] -> N ) ?   YES
       U.D.F. T [IN FACT]   FIX?    YES
       [IN FACT] (COND -- ((T --))) ->
                       (COND -- (T --))
       ITIMS [IN FACT] -> ITIMES ?    ...YES
       FACCT [IN FACT] -> FACT ?    ...YES
       9SUB1 [IN FACT] ->  ( SUB1 ?  NO
       U.B.A.
       (9SUB1 BROKEN)
       :
```

For most corrections, if you do not respond in a specified interval of time, DWIM automatically proceeds with the correction, so that you need intervene only when you do not approve. In the example, you responded to the first, second, and fifth questions; DWIM responded for you on the third and fourth.

DWIM uses ASKUSER for its interactions with you (see Chapter 26). Whenever an interaction is about to take place and you have typed ahead, ASKUSER types several bells to warn you to stop typing, then clears and saves the input buffers, restoring them after the interaction is complete. Thus if you typed ahead before a DWIM interaction, DWIM will not confuse your type-ahead with the answer to its question, nor will your type-ahead be lost. The bells are printed by the function PRINTBELLS, which can be advised or redefined for specialized applications, e.g. to flash the screen for a display terminal.

A great deal of effort has gone into making DWIM "smart", and experience with a large number of users indicates that DWIM works very well; DWIM seldom fails to correct an error you feel it should have, and almost never mistakenly corrects an error. However, it is important to note that even when DWIM *is* wrong, no harm is done: since an error had occurred, you would have had to intervene anyway if DWIM took no action. Thus, if DWIM mistakenly corrects an error, you simply interrupt or abort the computation, reverse the DWIM change using UNDO (see Chapter 13), and make the correction you would have had to make without DWIM. An exception is if DWIM's correction mistakenly caused a destructive computation to be initiated, and information was lost before you could interrupt. We have not yet had such an incident occur.

> (**DWIM** *X*)   [Function]
>
>> Used to enable/disable DWIM. If *X* is the symbol C, DWIM is enabled in CAUTIOUS mode, so that DWIM will ask you before making corrections. If *X* is T, DWIM is enabled in TRUSTING mode, so DWIM will make most corrections automatically. If *X* is NIL, DWIM is disabled. Medley initially has DWIM enabled in CAUTIOUS mode.
>>
>> DWIM returns CAUTIOUS, TRUSTING or NIL, depending to what mode it has just been put into.

For corrections to expressions typed in for immediate execution (typed into LISPX, Chapter 13), DWIM always acts as though it were in TRUSTING mode, i.e., no approval necessary. For certain types of corrections, e.g., run-on spelling corrections, 9-0 errors, etc., DWIM always acts like it was in CAUTIOUS mode, and asks for approval. In either case, DWIM always informs you of its action as described below.

Spelling Correction Protocol

One type of error that DWIM can correct is the misspelling of a function or a variable name. When an unbound symbol or undefined function error occurs, DWIM tries to correct the spelling of the bad symbol. If a symbol is found whose spelling is "close" to the offender, DWIM proceeds as follows:

If the correction occurs in the typed-in expression, DWIM prints =*CORRECT-SPELLING* and continues
evaluating the expression. For example:

```
←(SETQ FOO (IPLUSS 1 2))
        =IPLUS
        3
```

If the correction does not occur in type-in, DWIM prints

BAD-SPELLING [IN *FUNCTION-NAME*] -> *CORRECT-SPELLING*

The appearance of -> is to call attention to the fact that the user's function will be or has been
changed.

Then, if DWIM is in TRUSTING mode, it prints a carriage return, makes the correction, and continues
the computation. If DWIM is in CAUTIOUS mode, it prints a few spaces and ? and then wait for
approval. The user then has six options:

1. Type Y. DWIM types es, and proceeds with the correction.

2. Type N. DWIM types o, and does not make the correction.

3. Type ↑. DWIM does not make the correction, and furthermore guarantees that the error
    will not cause a break.

4. Type Control-E. For error correction, this has the same effect as typing N.

5. Do nothing. In this case DWIM waits for DWIMWAIT seconds, and if you have not
    responded, DWIM will type . . . followed by the default answer.

    The default on spelling corrections is determined by the value of the variable
    FIXSPELLDEFAULT, whose top level value is initially Y.

6. Type space or carriage-return. In this case DWIM will wait indefinitely. This option is
    intended for those cases where you want to think about your answer, and want to insure
    that DWIM does not get "impatient" and answer for you.

The procedure for spelling correction on other than Interlisp errors is analogous. If the correction is
being handled as type-in, DWIM prints = followed by the correct spelling, and returns it to the function
that called DWIM. Otherwise, DWIM prints the incorrect spelling, followed by the correct spelling.
Then, if DWIM is in TRUSTING mode, DWIM prints a carriage-return and returns the correct spelling.
Otherwise, DWIM prints a few spaces and a ? and waits for approval. You can then respond with Y, N,
Control-E, space, carriage return, or do nothing as described above.

The spelling corrector itself is not ERRORSET protected like the DWIM error correction routines.
Therefore, typing N and typing Control-E may have different effects when the spelling corrector is
called directly. The former simply instructs the spelling corrector to return NIL, and lets the calling

function decide what to do next; the latter causes an error which unwinds to the last ERRORSET, however far back that may be.

Parentheses Errors Protocol

When an unbound symbol or undefined error occurs, and the offending symbol contains 9 or 0, DWIM tries to correct errors caused by typing 9 for left parenthesis and 0 for right parenthesis. In these cases, the interaction with you is similar to that for spelling correction. If the error occurs in type-in, DWIM types =*CORRECTION*, and continues evaluating the expression. For example:

```
←(SETQ FOO 9IPLUS 1 2]
       = ( IPLUS
       3
```

If the correction does not occur in type-in, DWIM prints

BAD-ATOM [IN *FUNCTION-NAME*] -> *CORRECTION* ?

and then waits for approval. You then have the same six options as for spelling correction, except the waiting time is 3*DWIMWAIT seconds. If you type Y, DWIM operates as if it were in TRUSTING mode, i.e., it makes the correction and prints its message.

Actually, DWIM uses the value of the variables LPARKEY and RPARKEY to determine the corresponding lower case character for left and right parentheses. LPARKEY and RPARKEY are initially 9 and 0 respectively, but they can be reset for other keyboard layouts, e.g., on some terminals left parenthesis is over 8, and right parenthesis is over 9.

Undefined Function T Errors

When an undefined function error occurs, and the offending function is T, DWIM tries to correct certain types of parentheses errors involving a T clause in a conditional. DWIM recognizes errors of the following forms:

(COND --) (T --)	The T clause appears outside and immediately following the COND.
(COND -- (-- & (T --)))	The T clause appears inside a previous clause.
(COND -- ((T --)))	The T clause has an extra pair of parentheses around it.

For undefined function errors that are not one of these three types, DWIM takes no corrective action at all, and the error will occur.

If the error occurs in type-in, DWIM simply types `T  FIXED` and makes the correction. Otherwise if DWIM is in `TRUSTING` mode, DWIM makes the correction and prints the message:

```
[IN FUNCTION-NAME] {BAD-COND} ->
                {CORRECTED-COND}
```

If `DWIM` is in `CAUTIOUS` mode, DWIM prints

```
UNDEFINED FUNCTION T
        [IN FUNCTION-NAME]   FIX?
```

and waits for approval. You then have the same options as for spelling corrections and parenthesis errors. If you type `Y` or default, `DWIM` makes the correction and prints its message.

Having made the correction, `DWIM` must then decide how to proceed with the computation. In the first case, `(COND --) (T --)`, `DWIM` cannot know whether the `T` clause would have been executed if it had been inside of the `COND`. Therefore `DWIM` asks you `CONTINUE  WITH  T  CLAUSE` (with a default of `YES`). If you type `N`, `DWIM` continues with the form after the `COND`, i.e., the form that originally followed the `T` clause.

In the second case, `(COND -- (-- & (T --)))`, `DWIM` has a different problem. After moving the `T` clause to its proper place, `DWIM` must return as the value of `&` as the value of the `COND`. Since this value is no longer around, `DWIM` asks you `OK  TO  REEVALUATE` and then prints the expression corresponding to `&`. If you type `Y`, or default, `DWIM` continues by reevaluating `&`, otherwise `DWIM` aborts, and a `U.D.F.  T` error will then occur (even though the `COND` has in fact been fixed). If `DWIM` can determine for itself that the form can safely be reevaluated, it does not consult you before reevaluating. `DWIM` can do this if the form is atomic, or `CAR` of the form is a member of the list `OKREEVALST`, and each of the arguments can safely be reevaluated. For example, `(SETQ X (CONS (IPLUS Y Z) W))` is safe to reevaluate because `SETQ`, `CONS`, and `IPLUS` are all on `OKREEVALST`.

In the third case, `(COND  --  ((T  --)))`, there is no problem with continuation, so no further interaction is necessary.

DWIM Operation

Whenever the interpreter encounters an atomic form with no binding, or a non-atomic form `CAR` of which is not a function or function object, it calls the function `FAULTEVAL`. Similarly, when `APPLY` is given an undefined function, `FAULTAPPLY` is called. When `DWIM` is enabled, `FAULTEVAL` and `FAULTAPPLY` are redefined to first call the DWIM package, which tries to correct the error. If `DWIM` cannot decide how to fix the error, or you disapprove of `DWIM`'s correction (by typing `N`), or you type Control-E, then `FAULTEVAL` and `FAULTAPPLY` cause an error or break. If you type ↑ to `DWIM`, `DWIM` exits by performing `(RETEVAL 'FAULTEVAL '(ERROR!))`, so that an error will be generated at the position of the call to `FAULTEVAL`.

If DWIM can (and is allowed to) correct the error, it exits by performing RETEVAL of the corrected form, as of the position of the call to FAULTEVAL or FAULTAPPLY. Thus in the example at the beginning of the chapter, when DWIM determined that ITIMS was ITIMES misspelled, DWIM called RETEVAL with (ITIMES N (FACCT 9SUB1 N)). Since the interpreter uses the value returned by FAULTEVAL exactly as though it were the value of the erroneous form, the computation will thus proceed exactly as though no error had occurred.

In addition to continuing the computation, DWIM also repairs the cause of the error whenever possible; in the above example, DWIM also changed (with RPLACA) the expression (ITIMS N (FACCT 9SUB1 N)) that caused the error. Note that if your program had *computed* the form and called EVAL, it would not be possible to repair the cause of the error, although DWIM could correct the misspelling each time it occurred.

Error correction in DWIM is divided into three categories: unbound atoms, undefined CAR of form, and undefined function in APPLY. Assuming that the user approves DWIM's corrections, the action taken by DWIM for the various types of errors in each of these categories is summarized below.

DWIM Correction: Unbound Atoms

If DWIM is called as the result of an unbound atom error, it proceeds as follows:

1. If the first character of the unbound atom is ', DWIM assumes that you (intentionally) typed 'ATOM for (QUOTE ATOM) and makes the appropriate change. No message is typed, and no approval is requested.

   If the unbound atom is just ' itself, DWIM assumes you want the *next* expression quoted, e.g., (CONS X '(A B C)) will be changed to (CONS X (QUOTE (A B C))). Again no message will be printed or approval asked. If no expression follows the ', DWIM gives up.

   > Note: ' is normally defined as a read-macro character which converts 'FOO to (QUOTE FOO) on input, so DWIM will not see the ' in the case of expressions that are typed-in.

2. If CLISP (see Chapter 21) is enabled, and the atom is part of a CLISP construct, the CLISP transformation is performed and the result returned. For example, N-1 is transformed to (SUB1 N), and (... FOO_3 ...) is transformed into (... (SETQ FOO 3) ...).

3. If the atom contains an 9 (actually LPARKEY (see the DWIM Functions and Variables section below), DWIM assumes the **9** was intended to be a left parenthesis, and calls the editor to make appropriate repairs on the expression containing the atom. DWIM assumes that you did not notice the mistake, i.e., that the entire expression was affected by the missing left parenthesis. For example, if you type (SETQ X (LIST (CONS 9CAR Y) (CDR Z)) Y), the expression will be changed to (SETQ X (LIST (CONS (CAR Y) (CDR Z)) Y)). The 9 does not have to be the first character of the atom: DWIM will handle (CONS X9CAR Y) correctly.

4. If the atom contains a `0` (actually `RPARKEY`, see the DWIM Functions and Variables section below), DWIM assumes the `0` was intended to be a right parenthesis and operates as in the case above.

5. If the atom begins with a `7`, the `7` is treated as a `'`. For example, `7FOO` becomes `'FOO`, and then `(QUOTE FOO)`.

6. The expressions on `DWIMUSERFORMS` (see the `DWIMUSERFORMS` section below) are evaluated in the order that they appear. If any of these expressions returns a non-`NIL` value, this value is treated as the form to be used to continue the computation, it is evaluated and its value is returned by `DWIM`.

7. If the unbound atom occurs in a function, `DWIM` attempts spelling correction using the `LAMBDA` and `PROG` variables of the function as the spelling list.

8. If the unbound atom occurred in a type-in to a break, `DWIM` attempts spelling correction using the `LAMBDA` and `PROG` variables of the broken function as the spelling list.

9. Otherwise, `DWIM` attempts spelling correction using `SPELLINGS3` (see the Spelling Lists section below).

10. If all of the above fail, `DWIM` gives up.

Undefined CAR of Form

If DWIM is called as the result of an undefined `CAR` of form error, it proceeds as follows:

1. If `CAR` of the form is `T`, DWIM assumes a misplaced `T` clause and operates as described in the Undefined Function T Errors section above.

2. If `CAR` of the form is `F/L`, DWIM changes the "F/L" to "`FUNCTION(LAMBDA`". For example, `(F/L (Y) (PRINT (CAR Y)))` is changed to `(FUNCTION (LAMBDA (Y) (PRINT (CAR Y)))`. No message is printed and no approval requested. If you omit the variable list, DWIM supplies `(X)`, e.g., `(F/L (PRINT (CAR X)))` is changed to `(FUNCTION (LAMBDA (X) (PRINT (CAR X))))`. DWIM determines that you have supplied the variable list when more than one expression follows `F/L`, `CAR` of the first expression is not the name of a function, and every element in the first expression is atomic. For example, DWIM will supply `(X)` when correcting `(F/L (PRINT (CDR X)) (PRINT (CAR X)))`.

3. If `CAR` of the form is a CLISP word (`IF`, `FOR`, `DO`, `FETCH`, etc.), the indicated CLISP transformation is performed, and the result is returned as the corrected form. See Chapter 21.

4. If `CAR` of the form has a function definition, `DWIM` attempts spelling correction on `CAR` of the definition using as spelling list the value of `LAMBDASPLST`, initially (`LAMBDA NLAMBDA`).

5. If `CAR` of the form has an `EXPR` or `CODE` property, `DWIM` prints *CAR-OF-FORM* `UNSAVED`, performs an `UNSAVEDEF`, and continues. No approval is requested.

6. If `CAR` of the form has a `FILEDEF` property, the definition is loaded from a file (except when `DWIMIFY`ing). If the value of the property is atomic, the entire file is to be loaded. If the value is a list, `CAR` is the name of the file and `CDR` the relevant functions, and `LOADFNS` will be used. For both cases, *LDFLG* will be `SYSLOAD` (see Chapter 17). `DWIM` uses `FINDFILE` (Chapter 24), so that the file can be on any of the directories on `DIRECTORIES`, initially (`NIL NEWLISP LISP LISPUSERS`). If the file is found, DWIM types `SHALL I LOAD` followed by the file name or list of functions. If you approve, DWIM loads the function(s) or file, and continues the computation.

7. If CLISP is enabled, and `CAR` of the form is part of a CLISP construct, the indicated transformation is performed, e.g., (`N←N-1`) becomes (`SETQ N (SUB1 N)`).

8. If `CAR` of the form contains an `9`, `DWIM` assumes a left parenthesis was intended e.g., (`CONS9CAR X`).

9. If `CAR` of the form contains a `0`, `DWIM` assumes a right parenthesis was intended.

10. If `CAR` of the form is a list, DWIM attempts spelling correction on `CAAR` of the form using `LAMBDASPLST` as spelling list. If successful, `DWIM` returns the corrected expression itself.

11. The expressions on `DWIMUSERFORMS` are evaluated in the order they appear. If any returns a non-`NIL` value, this value is treated as the corrected form, it is evaluated, and `DWIM` returns its value.

12. Otherwise, `DWIM` attempts spelling correction using `SPELLINGS2` as the spelling list (see the Spelling Lists section below). When `DWIMIFY`ing, `DWIM` also attemps spelling correction on function names not defined but previously encountered, using `NOFIXFNSLST` as a spelling list (see Chapter 21).

13. If all of the above fail, DWIM gives up.

Undefined Function in APPLY

If `DWIM` is called as the result of an undefined function in `APPLY` error, it proceeds as follows:

1. If the function has a definition, DWIM attempts spelling correction on `CAR` of the definition using `LAMBDASPLST` as spelling list.

2. If the function has an `EXPR` or `CODE` property, DWIM prints *FN* `UNSAVED`, performs an `UNSAVEDEF` and continues. No approval is requested.

3. If the function has a property `FILEDEF`, DWIM proceeds as in case 6 of undefined `CAR` of form.

4. If the error resulted from type-in, and CLISP is enabled, and the function name contains a CLISP operator, DWIM performs the indicated transformation, e.g., type `FOO←(APPEND FIE FUM)`.

5. If the function name contains an `9`, DWIM assumes a left parenthesis was intended, e.g., `EDIT9FOO]`.

6. If the "function" is a list, DWIM attempts spelling correction on CAR of the list using LAMBDASPLST as spelling list.

7. The expressions on DWIMUSERFORMS are evaluated in the order they appear, and if any returns a non-NIL value, this value is treated as the function used to continue the computation, i.e., it will be applied to its arguments.

8. DWIM attempts spelling correction using SPELLINGS1 as the spelling list.

9. DWIM attempts spelling correction using SPELLINGS2 as the spelling list.

10. If all fail, DWIM gives up.

DWIMUSERFORMS

The variable DWIMUSERFORMS provides a convenient way of adding to the transformations that DWIM performs. For example, you might want to change atoms of the form $X to (QA4LOOKUP X). Before attempting spelling correction, but after performing other transformations (F/L, 9, 0, CLISP, etc.), DWIM evaluates the expressions on DWIMUSERFORMS in the order they appear. If any expression returns a non-NIL value, this value is treated as the transformed form to be used. If DWIM was called from FAULTEVAL, this form is evaluated and the resulting value is returned as the value of FAULTEVAL. If DWIM is called from FAULTAPPLY, this form is treated as a function to be applied to FAULTARGS, and the resulting value is returned as the value of FAULTAPPLY. If all of the expressions on DWIMUSERFORMS return NIL, DWIM proceeds as though DWIMUSERFORMS = NIL, and attempts spelling correction. Note that DWIM simply takes the value and returns it; the expressions on DWIMUSERFORMS are responsible for making any modifications to the original expression. The expressions on DWIMUSERFORMS should make the transformation permanent, either by associating it with FAULTX via CLISPTRAN, or by destructively changing FAULTX.

In order for an expression on DWIMUSERFORMS to be able to be effective, it needs to know various things about the context of the error. Therefore, several of DWIM's internal variables have been made SPECVARS (see Chapter 18) and are therefore "visible" to DWIMUSERFORMS. Below are a list of those variables that may be useful.

FAULTX   [Variable]

For unbound atom and undefined car of form errors, FAULTX is the atom or form. For undefined function in APPLY errors, FAULTX is the name of the function.

FAULTARGS   [Variable]

For undefined function in APPLY errors, FAULTARGS is the list of arguments. FAULTARGS may be modified or reset by expressions on DWIMUSERFORMS.

FAULTAPPLYFLG [Variable]

Value is `T` for undefined function in `APPLY` errors; `NIL` otherwise. The value of `FAULTAPPLYFLG` *after* an expression on `DWIMUSERFORMS` returns a non-`NIL` value determines how the latter value is to be treated. Following an undefined function in `APPLY` error, if an expression on `DWIMUSERFORMS` sets `FAULTAPPLYFLG` to `NIL`, the value returned is treated as a form to be evaluated, rather than a function to be applied.

`FAULTAPPLYFLG` is necessary to distinguish between unbound atom and undefined function in `APPLY` errors, since `FAULTARGS` may be `NIL` and `FAULTX` atomic in both cases.

TAIL [Variable]

For unbound atom errors, `TAIL` is the tail of the expression `CAR` of which is the unbound atom. `DWIMUSERFORMS` expression can replace the atom by another expression by performing (`/RPLACA TAIL` *EXPR*)

PARENT [Variable]

For unbound atom errors, `PARENT` is the form in which the unbound atom appears. `TAIL` is a tail of `PARENT`.

TYPE-IN? [Variable]

True if the error occurred in type-in.

FAULTFN [Variable]

Name of the function in which error occurred. `FAULTFN` is `TYPE-IN` when the error occurred in type-in, and `EVAL` or `APPLY` when the error occurred under an explicit call to `EVAL` or `APPLY`.

DWIMIFYFLG [Variable]

True if the error was encountered while `DWIMIFY`ing (as opposed to happening while running a program).

EXPR [Variable]

Definition of `FAULTFN`, or argument to `EVAL`, i.e., the superform in which the error occurs.

The initial value of `DWIMUSERFORMS` is ((`DWIMLOADFNS?`)). `DWIMLOADFNS?` is a function for automatically loading functions from files. If `DWIMLOADFNSFLG` is `T` (its initial value), and `CAR` of the form is the name of a function, and the function is contained on a file that has been noticed by the file package, the function is loaded, and the computation continues.

DWIM Functions and Variables

DWIMWAIT   [Variable]

Value is the number of seconds that DWIM will wait before it assumes that you are not going to respond to a question and uses the default response FIXSPELLDEFAULT.

DWIM operates by dismissing for 250 milliseconds, then checking to see if anything has been typed. If not, it dismisses again, etc. until DWIMWAIT seconds have elapsed. Thus, there will be a delay of at most 1/4 second before DWIM responds to your answer.

FIXSPELLDEFAULT   [Variable]

If approval is requested for a spelling correction, and you do not respond, defaults to value of FIXSPELLDEFAULT, initially Y. FIXSPELLDEFAULT is rebound to N when DWIMIFYing.

ADDSPELLFLG   [Variable]

If NIL, suppresses calls to ADDSPELL. Initially T.

NOSPELLFLG   [Variable]

If T, suppresses *all* spelling correction. If some other non-NIL value, suppresses spelling correction in programs but not type-in. NOSPELLFLG is initially NIL. It is rebound to T when compiling from a file.

RUNONFLG   [Variable]

If NIL, suppresses run-on spelling corrections. Initially NIL.

DWIMLOADFNSFLG   [Variable]

If T, tells DWIM that when it encounters a call to an undefined function contained on a file that has been noticed by the file package, to simply load the function. DWIMLOADFNSFLG is initially T (see above).

LPARKEY   [Variable]
RPARKEY   [Variable]

DWIM uses the value of the variables LPARKEY and RPARKEY (initially 9 and 0 respectively) to determine the corresponding lower case character for left and right parentheses. LPARKEY and RPARKEY can be reset for other keyboard layouts. For example, on some terminals left parenthesis is over 8, and right parenthesis is over 9.

OKREEVALST   [Variable]

The value of OKREEVALST is a list of functions that DWIM can safely reevaluate. If a form is atomic, or CAR of the form is a member of OKREEVALST, and each of the arguments can safely be reevaluated, then the form can be safely reevaluated. For example, (SETQ X (CONS (IPLUS Y Z) W)) is safe to reevaluate because SETQ, CONS, and IPLUS are all on OKREEVALST.

DWIMFLG [Variable]

> `DWIMFLG = NIL`, all `DWIM` operations are disabled. `(DWIM 'C)` and `(DWIM T)` set `DWIMFLG` to `T`; `(DWIM NIL)` sets `DWIMFLG` to `NIL`.

APPROVEFLG [Variable]

> `APPROVEFLG = T` if `DWIM` should ask the user for approval before making a correction that will modify the definition of one of his functions; `NIL` otherwise.

> When `DWIM` is put into `CAUTIOUS` mode with `(DWIM 'C)`, `APPROVEFLG` is set to `T`; for `TRUSTING` mode, `APPROVEFLG` is set to `NIL`.

LAMBDASPLST [Variable]

> `DWIM` uses the value of `LAMBDASPLST` as the spelling list when correcting "bad" function definitions. Initially `(LAMBDA NLAMBDA)`. You may wish to add to `LAMBDASPLST` if you elect to define new "function types" via an appropriate `DWIMUSERFORMS` entry. For example, the `QLAMBDA`s of SRI's QLISP are handled in this way.

Spelling Correction

The spelling corrector is given as arguments a misspelled word (word means symbol), a spelling list (a list of words), and a number: *XWORD*, *SPLST*, and *REL* respectively. Its task is to find that word on *SPLST* which is closest to *XWORD*, in the sense described below. This word is called a *respelling* of *XWORD*. *REL* specifies the minimum "closeness" between *XWORD* and a respelling. If the spelling corrector cannot find a word on *SPLST* closer to *XWORD* than *REL*, or if it finds two or more words equally close, its value is `NIL`, otherwise its value is the respelling. The spelling corrector can also be given an optional functional argument, *FN*, to be used for selecting out a subset of *SPLST*, i.e., only those members of *SPLST* that satisfy *FN* will be considered as possible respellings.

The exact algorithm for computing the spelling metric is described later, but briefly "closeness" is inversely proportional to the number of disagreements between the two words, and directly proportional to the length of the longer word. For example, `PRTTYPRNT` is "closer" to `PRETTYPRINT` than `CS` is to `CONS` even though both pairs of words have the same number of disagreements. The spelling corrector operates by proceeding down *SPLST*, and computing the closeness between each word and *XWORD*, and keeping a list of those that are closest. Certain differences between words are not counted as disagreements, for example a single transposition, e.g., `CONS` to `CNOS`, or a doubled letter, e.g., `CONS` to `CONSS`, etc. In the event that the spelling corrector finds a word on *SPLST* with *no* disagreements, it will stop searching and return this word as the respelling. Otherwise, the spelling corrector continues through the entire spelling list. Then if it has found one and only one "closest" word, it returns this word as the respelling. For example, if *XWORD* is `VONS`, the spelling corrector will probably return `CONS` as the respelling. However, if *XWORD* is `CONZ`, the spelling corrector will not be able to return a respelling, since `CONZ` is equally close to both `CONS` and `COND`. If the spelling corrector finds an acceptable respelling, it interacts with you as described earlier.

In the special case that the misspelled word contains one or more $s (escape), the spelling corrector searches for those words on *SPLST* that match *XWORD*, where a $ can match any number of characters (including 0), e.g., FOO$ matches FOO1 and FOO, but not NEWFOO. FOO matches all three. Both completion and correction may be involved, e.g. RPETTY$ will match PRETTYPRINT, with one mistake. The entire spelling list is always searched, and if more than one respelling is found, the spelling corrector prints AMBIGUOUS, and returns NIL. For example, CON$ would be ambiguous if both CONS and COND were on the spelling list. If the spelling corrector finds one and only one respelling, it interacts with you as described earlier.

For both spelling correction and spelling completion, regardless of whether or not you approve of the spelling corrector's choice, the respelling is moved to the front of *SPLST*. Since many respellings are of the type with no disagreements, this procedure has the effect of considerably reducing the time required to correct the spelling of frequently misspelled words.

Synonyms

Spelling lists also provide a way of defining synonyms for a particular context. If a dotted pair appears on a spelling list (instead of just an atom), CAR is interpreted as the correct spelling of the misspelled word, and CDR as the antecedent for that word. If CAR is *identical* with the misspelled word, the antecedent is returned without any interaction or approval being necessary. If the misspelled word *corrects* to CAR of the dotted pair, the usual interaction and approval will take place, and then the antecedent, i.e., CDR of the dotted pair, is returned. For example,you could make IFLG synonymous with CLISPIFTRANFLG by adding (IFLG . CLISPIFTRANFLG) to SPELLINGS3, the spelling list for unbound atoms. Similarly, you could make OTHERWISE mean the same as ELSEIF by adding (OTHERWISE . ELSEIF) to CLISPIFWORDSPLST, or make L be synonymous with LAMBDA by adding (L . LAMBDA) to LAMBDASPLST. You can also use L as a variable without confusion, since the association of L with LAMBDA occurs only in the appropriate context.

Spelling Lists

Any list of atoms can be used as a spelling list, e.g., BROKENFNS, FILELST, etc. Various system packages have their own spellings lists, e.g., LISPXCOMS, CLISPFORWORDSPLST, EDITCOMSA, etc. These are documented under their corresponding sections, and are also indexed under "spelling lists." In addition to these spelling lists, the system maintains, i.e., automatically adds to, and occasionally prunes, four lists used solely for spelling correction: SPELLINGS1, SPELLINGS2, SPELLINGS3, and USERWORDS. These spelling lists are maintained *only* when ADDSPELLFLG is non-NIL. ADDSPELLFLG is initially T.

SPELLINGS1 [Variable]

SPELLINGS1 is a list of functions used for spelling correction when an input is typed in apply format, and the function is undefined, e.g., EDTIF(FOO). SPELLINGS1 is initialized to contain DEFINEQ, BREAK, MAKEFILE, EDITF, TCOMPL, LOAD, etc. Whenever LISPX is given an input in apply format, i.e., a function and arguments, the name of the function is added to SPELLINGS1 if the function has a definition.

For example, typing `CALLS(EDITF)` will cause `CALLS` to be added to `SPELLINGS1`. Thus if you typed `CALLS(EDITF)` and later typed `CALLLS(EDITV)`, since `SPELLINGS1` would then contain `CALLS`, DWIM would be successful in correcting `CALLLS` to `CALLS`.

SPELLINGS2 [Variable]

> `SPELLINGS2` is a list of functions used for spelling correction for all other undefined functions. It is initialized to contain functions such as `ADD1`, `APPEND`, `COND`, `CONS`, `GO`, `LIST`, `NCONC`, `PRINT`, `PROG`, `RETURN`, `SETQ`, etc. Whenever `LISPX` is given a non-atomic form, the name of the function is added to `SPELLINGS2`. For example, typing `(RETFROM (STKPOS (QUOTE FOO) 2))` to a break would add `RETFROM` to `SPELLINGS2`. Function names are also added to `SPELLINGS2` by `DEFINE`, `DEFINEQ`, `LOAD` (when loading compiled code), `UNSAVEDEF`, `EDITF`, and `PRETTYPRINT`.

SPELLINGS3 [Variable]

> `SPELLINGS3` is a list of words used for spelling correction on all unbound atoms. `SPELLINGS3` is initialized to `EDITMACROS`, `BREAKMACROS`, `BROKENFNS`, and `ADVISEDFNS`. Whenever `LISPX` is given an atom to evaluate, the name of the atom is added to `SPELLINGS3` if the atom has a value. Atoms are also added to `SPELLINGS3` whenever they are edited by `EDITV`, and whenever they are set via `RPAQ` or `RPAQQ`. For example, when a file is loaded, all of the variables set in the file are added to `SPELLINGS3`. Atoms are also added to `SPELLINGS3` when they are set by a `LISPX` input, e.g., typing `(SETQ FOO (REVERSE (SETQ FIE ...)))` will add both `FOO` and `FIE` to `SPELLINGS3`.

USERWORDS [Variable]

> `USERWORDS` is a list containing both functions and variables that you have *referred* to, e.g., by breaking or editing. `USERWORDS` is used for spelling correction by `ARGLIST`, `UNSAVEDEF`, `PRETTYPRINT`, `BREAK`, `EDITF`, `ADVISE`, etc. `USERWORDS` is initially `NIL`. Function names are added to it by `DEFINE`, `DEFINEQ`, `LOAD`, (when loading compiled code, or loading exprs to property lists) `UNSAVEDEF`, `EDITF`, `EDITV`, `EDITP`, `PRETTYPRINT`, etc. Variable names are added to `USERWORDS` at the same time as they are added to `SPELLINGS3`. In addition, the variable `LASTWORD` is always set to the last word added to `USERWORDS`, i.e., the last function or variable referred to by the user, and the respelling of `NIL` is defined to be the value of `LASTWORD`. Thus, if you had just defined a function, you can then prettyprint it by typing `PP()`.

Each of the above four spelling lists are divided into two sections separated by a special marker (the value of the variable `SPELLSTR1`). The first section contains the "permanent" words; the second section contains the temporary words. New words are added to the corresponding spelling list at the front of its temporary section (except that functions added to `SPELLINGS1` or `SPELLINGS2` by `LISPX` are always added to the end of the permanent section. If the word is already in the temporary section, it is moved to the front of that section; if the word is in the permanent section, no action is taken. If the length of the temporary section then exceeds a specified number, the last (oldest) word in the temporary section is forgotten, i.e., deleted. This procedure prevents the spelling lists from becoming cluttered with unimportant words that are no longer being used, and thereby slowing down spelling

correction time. Since the spelling corrector usually moves each word selected as a respelling to the front of its spelling list, the word is thereby moved into the permanent section. Thus once a word is misspelled and corrected, it is considered important and will never be forgotten.

The spelling correction algorithm will not alter a spelling list unless it contains the special marker (the value of `SPELLSTR1`). This provides a way to ensure that a spelling list will not be altered.

#SPELLINGS1	[Variable]
#SPELLINGS2	[Variable]
#SPELLINGS3	[Variable]
#USERWORDS	[Variable]

> The maximum length of the temporary section for `SPELLINGS1`, `SPELLINGS2`, `SPELLINGS3` and `USERWORDS` is given by the value of `#SPELLINGS1`, `#SPELLINGS2`, `#SPELLINGS3`, and `#USERWORDS`, initialized to 30, 30, 30, and 60 respectively.

> You can alter these values to modify the performance behavior of spelling correction.

Generators for Spelling Correction

For some applications, it is more convenient to *generate* candidates for a respelling one by one, rather than construct a complete list of all possible candidates, e.g., spelling correction involving a large directory of files, or a natural language data base. For these purposes, *SPLST* can be an array (of any size). The first element of this array is the generator function, which is called with the array itself as its argument. Thus the function can use the remainder of the array to store "state" information, e.g., the last position on a file, a pointer into a data structure, etc. The value returned by the function is the next candidate for respelling. If `NIL` is returned, the spelling "list" is considered to be exhausted, and the closest match is returned.   If a candidate is found with no disagreements, it is returned immediately without waiting for the "list" to exhaust.

SPLST can also be a generator, i.e. the value of the function `GENERATOR` (Chapter 11). The generator *SPLST* will be started up whenever the spelling corrector needs the next candidate, and it should return candidates via the function `PRODUCE`. For example, the following could be used as a "spelling list" which effectively contains all functions in the system:

```
[GENERATOR
        (MAPATOMS (FUNCTION (LAMBDA (X) (if (GETD X) then (PRODUCE
X]
```

Spelling Corrector Algorithm

The basic philosophy of `DWIM` spelling correction is to count the number of disagreements between two words, and use this number divided by the length of the longer of the two words as a measure of their relative disagreement. One minus this number is then the relative agreement or closeness. For example, `CONS` and `CONX` differ only in their last character. Such substitution errors count as one disagreement, so that the two words are in 75% agreement. Most calls to the spelling corrector specify a relative agreement of 70, so that a single substitution error is permitted in words of four characters

or longer. However, spelling correction on shorter words is possible since certain types of differences such as single transpositions are not counted as disagreements. For example, AND and NAD have a relative agreement of 100. Calls to the spelling corrector from DWIM use the value of FIXSPELLREL, which is initially 70. Note that by setting FIXSPELLREL to 100, only spelling corrections with "zero" mistakes, will be considered, e.g., transpositions, double characters, etc.

The central function of the spelling corrector is CHOOZ. CHOOZ takes as arguments: a word, a minimum relative agreement, a spelling list, and an optional functional argument, *XWORD*, *REL*, *SPLST*, and *FN* respectively.

CHOOZ proceeds down *SPLST* examining each word. Words not satisfying *FN* (if *FN* is non-NIL), or those obviously too long or too short to be sufficiently close to *XWORD* are immediately rejected. For example, if *REL* = 70, and *XWORD* is 5 characters long, words longer than 7 characters will be rejected.

Special treatment is necessary for words shorter than *XWORD*, since doubled letters are not counted as disagreements. For example, CONNSSS and CONS have a relative agreement of 100. CHOOZ handles this by counting the number of doubled characters in *XWORD* before it begins scanning *SPLST*, and taking this into account when deciding whether to reject shorter words.

If *TWORD*, the current word on *SPLST*, is not rejected, CHOOZ computes the number of disagreements between it and *XWORD* by calling a subfunction, SKOR.

SKOR operates by scanning both words from left to right one character at a time. SKOR operates on the list of character codes for each word. This list is computed by CHOOZ before calling SKOR. Characters are considered to agree if they are the same characters or appear on the same key (i.e., a shift mistake). The variable SPELLCASEARRAY is a CASEARRAY which is used to determine equivalence classes for this purpose. It is initialized to equivalence lowercase and upper case letters, as well as the standard key transitions: for example, 1 with !, 3 with #, etc.

If the first character in *XWORD* and *TWORD* do *not* agree, SKOR checks to see if either character is the same as one previously encountered, and not accounted-for at that time. (In other words, transpositions are not handled by lookahead, but by *lookback*.) A displacement of two or fewer positions is counted as a tranposition; a displacement by more than two positions is counted as a disagreement.In either case, both characters are now considered as accounted for and are discarded, and SKORing continues.

If the first character in *XWORD* and *TWORD* do not agree, and neither agree with previously unaccounted-for characters, and *TWORD* has more characters remaining than *XWORD*, SKOR removes and saves the first character of *TWORD*, and continues by comparing the rest of *TWORD* with *XWORD* as described above. If *TWORD* has the same or fewer characters remaining than *XWORD*, the procedure is the same except that the character is removed from *XWORD*. In this case, a special check is first made to see if that character is equal to the *previous* character in *XWORD*, or to the *next* character in *XWORD*, i.e., a double character typo, and if so, the character is considered accounted-for, and not counted as a disagreement. In this case, the "length" of *XWORD* is also decremented. Otherwise making *XWORD*

sufficiently long by adding double characters would make it be arbitrarily close to *TWORD*, e.g., XXXXXX would correct to PP.

When SKOR has finished processing both *XWORD* and *TWORD* in this fashion, the value of SKOR is the number of unaccounted-for characters, plus the number of disagreements, plus the number of tranpositions, with two qualifications:

1. If both *XWORD* and *TWORD* have a character unaccounted-for in the same position, the two characters are counted only once, i.e., substitution errors count as only one disagreement, not two

2. If there are no unaccounted-for characters and no disagreements, transpositions are not counted.

This permits spelling correction on very short words, such as edit commands, e.g., XRT–>XTR. Transpositions are also not counted when FASTYPEFLG = T, for example, IPULX and IPLUS will be in 80% agreement with FASTYPEFLG = T, only 60% with FASTYPEFLG = NIL. The rationale behind this is that transpositions are much more common for fast typists, and should not be counted as disagreements, whereas more deliberate typists are not as likely to combine tranpositions and other mistakes in a single word, and therefore can use more conservative metric. FASTYPEFLG is initially NIL.

Spelling Corrector Functions and Variables

(**ADDSPELL** *X SPLST N*) [Function]

> Adds *X* to one of the spelling lists as determined by the value of *SPLST*:

NIL	Adds *X* to USERWORDS and to SPELLINGS2. Used by DEFINEQ.
0	Adds *X* to USERWORDS. Used by LOAD when loading EXPRs to property lists.
1	Adds *X* to SPELLINGS1 (at end of permanent section). Used by LISPX.
2	Adds *X* to SPELLINGS2 (at end of permanent section). Used by LISPX.
3	Adds *X* to USERWORDS and SPELLINGS3.
a spelling list	If *SPLST* is a spelling list, *X* is added to it. In this case, *N* is the (optional) length of the temporary section.
	If *X* is already on the spelling list, and in its temporary section, ADDSPELL moves *X* to the front of that section.

> ADDSPELL sets LASTWORD to *X* when *SPLST* = NIL, 0 or 3.

> If *X* is not a symbol, ADDSPELL takes no action.

Note that the various systems calls to ADDSPELL, e.g., from DEFINE, EDITF, LOAD, etc., can all be suppressed by setting or binding ADDSPELLFLG to NIL (see the DWIM Functions and Variables section above).

(**MISSPELLED?** *XWORD REL SPLST FLG TAIL FN*)   [Function]

If *XWORD* = NIL or $ (<esc>), MISSPELLED? prints = followed by the value of LASTWORD, and returns this as the respelling, without asking for approval. Otherwise, MISSPELLED? checks to see if *XWORD* is really misspelled, i.e., if *FN* applied to *XWORD* is true, or *XWORD* is already contained on *SPLST*. In this case, MISSPELLED? simply returns *XWORD*. Otherwise MISSPELLED? computes and returns (FIXSPELL *XWORD REL SPLST FLG TAIL FN*).

(**FIXSPELL** *XWORD REL SPLST FLG TAIL FN TIEFLG DONTMOVETOPFLG*)   [Function]

The value of FIXSPELL is either the respelling of or NIL. If for some reason itself is on , then FIXSPELL aborts and calls ERROR!. If there is a possibility that is spelled correctly, MISSPELLED? should be used instead of FIXSPELL. FIXSPELL performs all of the interactions described earlier, including requesting your approval if necessary.

If *XWORD* = NIL or $ (escape), the respelling is the value of LASTWORD, and no approval is requested.

If *XWORD* contains lowercase characters, and the corresponding uppercase word is correct, i.e. on *SPLST* or satisfies *FN*, the uppercase word is returned and no interaction is performed. If FIXSPELL.UPPERCASE.QUIET is NIL (the default), a warning "=XX" is printed when coercing from "xx" to "XX". If FIXSPELL.UPPERCASE.QUIET is non-NIL, no warning is given.

If *REL* = NIL, defaults to the value of FIXSPELLREL (initially 70).

If *FLG* = NIL, the correction is handled in type-in mode, i.e., approval is never requested, and *XWORD* is not typed. If *FLG* = T, *XWORD* is typed (before the =) and approval is requested if APPROVEFLG = T. If *FLG* = NO-MESSAGE, the correction is returned with no further processing. In this case, a run-on correction will be returned as a dotted pair of the two parts of the word, and a synonym correction as a list of the form (*WORD1 WORD2*), where *WORD1* is (the corrected version of) *XWORD*, and *WORD2* is the synonym. The effect of the function CHOOZ can be obtained by calling FIXSPELL with *FLG* = NO-MESSAGE.

If *TAIL* is not NIL, and the correction is successful, CAR of *TAIL* is replaced by the respelling (using /RPLACA).

FIXSPELL will attempt to correct misspellings caused by running two words together, if the global variable RUNONFLG is non-NIL (default is NIL). In this case, approval is always requested. When a run-on error is corrected, CAR of *TAIL* is replaced by the two words, and the value of FIXSPELL is the first one. For example, if FIXSPELL is called to correct the edit command (MOVE TO AFTERCOND 3 2) with *TAIL* = (AFTERCOND 3 2), *TAIL* would be changed to (AFTER COND 2 3), and FIXSPELL would return AFTER (subject to yourapproval where necessary). If *TAIL* = T, FIXSPELL will also perform run-

on corrections, returning a dotted pair of the two words in the event the correction is of this type.

If *TIEFLG* = NIL and a tie occurs, i.e., more than one word on *SPLST* is found with the same degree of "closeness", FIXSPELL returns NIL, i.e., no correction. If *TIEFLG* = PICKONE and a tie occurs, the first word is taken as the correct spelling. If *TIEFLG* = LIST, the value of FIXSPELL is a list of the respellings (even if there is only one), and FIXSPELL will not perform any interaction with you, nor modify *TAIL*, the idea being that the calling program will handle those tasks. Similarly, if *TIEFLG* = EVERYTHING, a list of all candidates whose degree of closeness is above *REL* will be returned, regardless of whether some are better than others. No interaction will be performed.

If *DONTMOVETOPFLG* = T and a correction occurs, it will *not* be moved to the front of the spelling list. Also, the spelling list will not be altered unless it contains the special marker used to separate the temporary and perminant parts of the system spelling lists (the value of SPELLSTR1).

(**FNCHECK** *FN NOERRORFLG SPELLFLG PROPFLG TAIL*)   [Function]

The task of FNCHECK is to check whether *FN* is the name of a function and if not, to correct its spelling. If *FN* is the name of a function or spelling correction is successful, FNCHECK adds the (corrected) name of the function to USERWORDS using ADDSPELL, and returns it as its value.

Since FNCHECK is called by many low level functions such as ARGLIST, UNSAVEDEF, etc., spelling correction only takes place when DWIMFLG = T, so that these functions can operate in a small Interlisp system which does not contain DWIM.

NOERRORFLG informs FNCHECK whether or not the calling function wants to handle the unsuccessful case: if *NOERRORFLG* is T, FNCHECK simply returns NIL, otherwise it prints fn NOT A FUNCTION and generates a non-breaking error.

If *FN* does not have a definition, but does have an EXPR property, then spelling correction is not attempted. Instead, if *PROPFLG* = T, *FN* is considered to be the name of a function, and is returned. If *PROPFLG* = NIL, *FN* is *not* considered to be the name of a function, and NIL is returned or an error generated, depending on the value of *NOERRORFLG*.

FNCHECK calls MISSPELLED? to perform spelling correction, so that if *FN* = NIL, the value of LASTWORD will be returned. *SPELLFLG* corresponds to MISSPELLED?'s fourth argument, *FLG*. If *SPELLFLG* = T, approval will be asked if DWIM was enabled in CAUTIOUS mode, i.e., if APPROVEFLG = T. *TAIL* corresponds to the fifth argument to MISSPELLED?.

FNCHECK is currently used by ARGLIST, UNSAVEDEF, PRETTYPRINT, BREAK0, BREAKIN, ADVISE, and CALLS. For example, BREAK0 calls FNCHECK with *NOERRORFLG* = T since if FNCHECK cannot produce a function, BREAK0 wants to define a dummy one. CALLS however calls FNCHECK with *NOERRORFLG* = NIL, since it cannot operate without a function.

Many other system functions call MISSPELLED? or FIXSPELL directly. For example, BREAK1 calls FIXSPELL on unrecognized atomic inputs before attempting to evaluate them, using as a spelling list a list of all break commands. Similarly, LISPX calls FIXSPELL on atomic inputs using a list of all LISPX commands. When UNBREAK is given
the name of a function that is not broken, it calls FIXSPELL with two different spelling lists, first with BROKENFNS, and if that fails, with USERWORDS. MAKEFILE calls MISSPELLED? using FILELST as a spelling list. Finally, LOAD, BCOMPL, BRECOMPILE, TCOMPL, and RECOMPILE all call MISSPELLED? if their input file(s) won't open.

20. CLISP

The syntax of Lisp is very simple. It can be described concisely, but it makes Lisp difficult to read and write without tools. Unlike many languages, there are no reserved words in Lisp such as IF, THEN, FOR, DO, etc., nor reserved characters like +, -, =, ←, etc. The only components of the language are atoms and delimiters. This eliminates the need for parsers and precedence rules, and makes Lisp programs easy to mainpuilate. For example, a Lisp interpreter can be written in one or two pages of Lisp code. This makes Lisp the most suitable programming language for writing programs that deal with other programs as data.

Human language is based on more complicated structures and relies more on special words to carry the meaning. The definiton of the factorial function looks like this in Lisp:

```
(COND ((ZEROP N) 1) (T (TIMES N (FACTORIAL ((SUB1 N))))))
```

This definition is easy to read for a machine but difficult to read for a human. CLISP is designed to make Interlisp programs easier to read and write. CLISP does this by translating various operators, conditionals, and iterative statements to Interlisp. For example, factorial can be written in CLISP:

```
(IF N = 0 THEN 1 ELSE N*(FACTORIAL N-1))
```

CLISP will translate this expression to the form in the example above. The translation will take place when the form is read so there are no performance penalties.

You should view CLISP as a shothand for produceing Lisp programs. CLISP makes a program easy to read and sometimes more compact.

CLISP is implemented via the error correction machinery in Interlisp (see Chapter 20). Any expression that Interlisp thinks is well-formed will never be seen by CLISP This means that interpreted programs that do not use CLISP constructs do not pay for its availability by slower execution time. In fact, the Interlisp interpreter does not know about CLISP at all. When the interpreter finds an error it calls an error routine which in turn invokes the Do-What-I-Mean (DWIM) analyzer. The DWIM analyzer knows how to deal with CLISP expressions. If the expression in question turns out to be a CLISP construct, the translated form is returned to the interpreter. In addition, the original CLISP expression is modified so that it *becomes* the correctly translated Interlisp form. In this way, the analysis and translation are done only once.

Integrating CLISP into Medley makes possible Do-What-I-Mean features for CLISP constructs as well as for pure Lisp expressions. For example, if you have defined a function named GET-PARENT, CLISP would know not to attempt to interpret the form (GET-PARENT) as an arithmetic infix operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If you mistakenly write (GET-PRAENT), CLISP would know you meant (GET-PARENT), and not (DIFFERENCE GET PRAENT), by using the information that PARENT is not the name of a variable, and that GET-PARENT is the name of a user function whose spelling is "very close" to that of GET-

PRAENT. Similarly, by using information about the program's environment not readily available to a preprocessor, CLISP can successfully resolve the following sorts of ambiguities:

1. (LIST X*FACT N), where FACT is the name of a variable, means (LIST (X*FACT) N).

2. (LIST X*FACT N), where FACT is *not* the name of a variable but instead is the name of a function, means (LIST X*(FACT N)), i.e., N is FACT's argument.

3. (LIST X*FACT(N)), FACT the name of a function (and not the name of a variable), means (LIST X*(FACT N)).

4. Cases 1, 2 and 3 with FACT misspelled!

The first expression is correct both from the standpoint of CLISP syntax and semantics so the change would be made notification. In the other cases, you would be informed or consulted about what was taking place. For example, suppose you write the expression (LIST X*FCCT N). Assume also that there was both a function named FACT and a variable named FCT.

1. You will first be asked if FCCT is a misspelling of FCT. If you say YES, the expression will be interpreted as (LIST (X*FCT) N). If you say NO, you will be asked if FCCT was a misspelling of FACT, i.e., if you intended X*FCCT N to mean X*(FACT N).

2. If you say YES to this question, the indicated transformation will be performed. If you say NO, the system will ask if X*FCCT should be treated as CLISP, since FCCT is not the name of a (bound) variable.

3. If you say YES, the expression will be transformed, if NO, it will be left alone, i.e., as (LIST X*FCCT N). Note that we have not even considered the case where X*FCCT is itself a misspelling of a variable name, e.g., a variable named XFCT (as with GET-PRAENT). This sort of transformation will be considered after you said NO to X*FCCT N -> X*(FACT N).

The question of whether X*FCCT should be treated as CLISP is important because Interlisp users may have programs that employ identifiers containing CLISP operators. Thus, if CLISP encounters the expression A/B in a context where either A or B are not the names of variables, it will ask you if A/B is intended to be CLISP, in case you really do have a free variable named A/B.

> Note: Through the discussion above, we speak of CLISP or DWIM asking you. Actually, if you typed in the expression in question for immediate execution, you are simply informed of the transformation, on the grounds that you would prefer an occasional misinterpretation rather than being continuously bothered, especially since you can always retype what you intended if a mistake occurs, and ask the programmer's assistant to UNDO the effects of the mistaken operations if necessary. For transformations on expressions in your programs, you can tell CLISP whether you wish to operate in CAUTIOUS or TRUSTING mode. In the former case (most typical) you will be asked to approve transformations, in the latter, CLISP will operate as it does on type-in, i.e., perform the transformation after informing you.

CLISP can also handle parentheses errors caused by typing 8 or 9 for (or). (On most terminals, 8 and 9 are the lowercase characters for (and), i.e., (and 8 appear on the same key, as do) and 9.) For example, if you write N*8FACTORIAL N-1, the parentheses error can be detected and fixed before the infix operator * is converted to the Interlisp function TIMES. CLISP is able to distinguish this situation from cases like N*8*X meaning (TIMES N 8 X), or N*8X, where 8X is the name of a variable, again by using information about the programming environment. In fact, by integrating CLISP with DWIM, CLISP has been made sufficiently tolerant of errors that almost everything can be misspelled! For example, CLISP can successfully translate the definition of FACTORIAL:

        (IFF N = 0 THENN1 ESLE N*8FACTTORIALNN-1)

to the corresponding COND, while making five spelling corrections and fixing the parenthesis error. CLISP also contains a facility for converting from Interlisp back to CLISP, so that after running the above incorrect definition of FACTORIAL, you could "clispify" the now correct version to obtain (IF N = 0 THEN 1 ELSE N*(FACTORIAL N-1)).

This sort of robustness prevails throughout CLISP. For example, the iterative statement permits you to say things like:

        (FOR OLD X FROM M TO N DO (PRINT X) WHILE (PRIMEP X))

However, you can also write OLD (X←M), (OLD X←M), (OLD (X←M)), permute the order of the operators, e.g., (DO PRINT X TO N FOR OLD X←M WHILE PRIMEP X), omit either or both sets of parentheses, misspell any or all of the operators FOR, OLD, FROM, TO, DO, or WHILE, or leave out the word DO entirely! And, of course, you can also misspell PRINT, PRIMEP, M or N! In this example, the only thing you could not misspell is the first X, since it specifies the *name* of the variable of iteration. The other two instances of X could be misspelled.

CLISP is well integrated into Medley. For example, the above iterative statement translates into an equivalent Interlisp form using PROG, COND, GO, etc. When the interpreter subsequently encounters this CLISP expression, it automatically obtains and evaluates the translation. Similarly, the compiler "knows" to compile the translated form. However, if you PRETTYPRINT your program, PRETTYPRINT "knows" to print the original CLISP at the corresponding point in your function. Similarly, when you edit your program, the editor keeps the translation invisible to you. If you modify the CLISP, the translation is automatically discarded and recomputed the next time the expression is evaluated.

In short, CLISP is not a language at all, but rather a system. It plays a role analagous to that of the programmer's assistant (Chapter 13). Whereas the programmer's assistant is an invisible intermediary agent between your console requests and the Interlisp executive, CLISP sits between your programs and the Interlisp interpreter.

Only a small effort has been devoted to defining the core syntax of CLISP. Instead, most of the effort has been concentrated on providing a facility which "makes sense" out of the input expressions using context information as well as built-in and acquired information about user and system programs. It has been said that communication is based on the intention of the speaker to produce an effect in the

recipient. CLISP operates under the assumption that what you say is *intended* to represent a meaningful operation, and therefore tries very hard to make sense out of it. The motivation behind CLISP is not to provide you with many different ways of saying the same thing, but to enable you to worry less about the *syntactic* aspects of your communication with the system. In other words, it gives you a new degree of freedom by permitting you to concentrate more on the problem at hand, rather than on translation into a formal and unambiguous language.

DWIM and CLISP are invoked on iterative statements because `CAR` of the iterative statement is not the name of a function, and hence generates an error. If you define a function by the same name as an i.s. operator, e.g., `WHILE`, `TO`, etc., the operator will no longer have the CLISP interpretation when it appears as `CAR` of a form, although it will continue to be treated as an i.s. operator if it appears in the interior of an i.s. To alert you, a warning message is printed, e.g., `(WHILE DEFINED, THEREFORE DISABLED IN CLISP)`.

CLISP Interaction with User

Syntactically and semantically well formed CLISP transformations are always performed without informing you. Other CLISP transformations described in the previous section, e.g., misspellings of operands, infix operators, parentheses errors, unary minus - binary minus errors, all follow the same protocol as other DWIM transformations (Chapter 19). That is, if DWIM has been enabled in `TRUSTING` mode, or the transformation is in an expression you typed in for immediate execution, your approval is not requested, but you are informed. However, if the transformation involves a user program, and DWIM was enabled in `CAUTIOUS` mode, you will be asked to approve. If you say `NO`, the transformation is not performed. Thus, in the previous section, phrases such as "one of these (transformations) succeeds" and "the transformation `LAST-ELL -> LAST-EL` would be found" etc., all mean if you are in `CAUTIOUS` mode and the error is in a program, the corresponding transformation will be performed only if you approve (or defaults by not responding). If you say `NO`, the procedure followed is the same as though the transformation had not been found. For example, if `A*B` appears in the function `FOO`, and B is not bound (and no other transformations are found) you would be asked `A*B [IN FOO]   TREAT AS CLISP ?` (The waiting time on such interactions is three times as long as for simple corrections, i.e., `3*DWIMWAIT`).

In certain situations, DWIM asks for approval even if DWIM is enabled in `TRUSTING` mode. For example, you are always asked to approve a spelling correction that might also be interpreted as a CLISP transformation, as in `LAST-ELL -> LAST-EL`.

If you approved, `A*B` would be transformed to `(ITIMES A B)`, which would then cause a `U.B.A.B.` error in the event that the program was being run (remember the entire discussion also applies to DWIMifying). If you said `NO`, `A*B` would be left alone.

If the value of `CLISPHELPFLG = NIL` (initally `T`), you will not be asked to approve any CLISP transformation. Instead, in those situations where approval would be required, the effect is the same as though you had been asked and said `NO`.

CLISP Character Operators

CLISP recognizes a number of special characters operators, both prefix and infix, which are translated into common expressions. For example, the character + is recognized to represent addition, so CLISP translates the symbol A+B to the form (IPLUS A B). Note that CLISP is invoked, and this translation is made, only if an error occurs, such as an unbound atom error or an undefined function error for the perfectly legitamate symbol A+B. Therefore you may choose not to use these facilities with no penalty, similar to other CLISP facilities.

You have a lot of flexability in using CLISP character operators. A list can always be substituted for a symbol, and vice versa, without changing the interpretation of a phrase. For example, if the value of (FOO X) is A, and the value of (FIE Y) is B, then (LIST (FOO X)+(FIE Y)) has the same value as (LIST A+B). Note that the first expression is a list of *four* elements: the atom "LIST", the list "(FOO X)", the atom "+", and the list "(FIE X)", whereas the second expression, (LIST A+B), is a list of only *two* elements: the symbol "LIST" and the symbol "A+B". Since (LIST (FOO X)+(FIE Y)) is indistinguishable from (LIST (FOO X) + (FIE Y)) because spaces before or after parentheses have no effect on the Interlisp READ program, to be consistent, extra spaces have no effect on atomic operands either. In other words, CLISP will treat (LIST A+ B), (LIST A +B), and (LIST A + B) the same as (LIST A+B).

> Note: CLISP does not use its own special READ program because this would require you to explicitly identify CLISP expressions, instead of being able to intermix Interlisp and CLISP.

+   [CLISP Operator]
-   [CLISP Operator]
*   [CLISP Operator]
/   [CLISP Operator]
↑   [CLISP Operator]

> CLISP recognizes +, -, *, /, and ↑ as the normal arithmetic infix operators. The - is also recognized as the prefix operator, unary minus. These are converted to PLUS, DIFFERENCE (or in the case of unary minus, MINUS), TIMES, QUOTIENT, and EXPT.

> Normally, CLISP uses the "generic" arithmetic functions PLUS, TIMES, etc. CLISP contains a facility for declaring which type of arithmetic is to be used, either by making a global declaration, or by separate declarations about individual functions or variables.

> The usual precedence rules apply (although you can easily change them), i.e., * has higher precedence than + so that A+B*C is the same as A+(B*C), and both * and / are lower than ↑ so that 2*X↑2 is the same as 2*(X↑2). Operators of the same precedence group from left to right, e.g., A/B/C is equivalent to (A/B)/C. Minus is binary whenever possible, i.e., except when it is the first operator in a list, as in (-A) or (-A), or when it immediately follows another operator, as in A*-B. Note that grouping with parentheses can always be used to override the normal precedence grouping, or when you are not sure how a particular expression will parse. The complete order of precedence for CLISP operators is given below.

Note that + in front of a number will disappear when the number is read, e.g., `(FOO X +2)` is indistinguishable from `(FOO X 2)`. This means that `(FOO X +2)` will not be interpreted as CLISP, or be converted to `(FOO (IPLUS X 2))`. Similarly, `(FOO X -2)` will not be interpreted the same as `(FOO X-2)`. To circumvent this, always type a space between the + or – and a number if an infix operator is intended, e.g., write `(FOO X + 2)`.

=	[CLISP Operator]
GT	[CLISP Operator]
LT	[CLISP Operator]
GE	[CLISP Operator]
LE	[CLISP Operator]

These are infix operators for "Equal", "Greater Than", "Less Than", "Greater Than or Equal", and "Less Than or Equal".

`GT`, `LT`, `GE`, and `LE` are all affected by the same declarations as + and `*`, with the initial default to use `GREATERP` and `LESSP`.

Note that only single character operators, e.g., +, ←, =, etc., can appear in the *interior* of an atom. All other operators must be set off from identifiers with spaces. For example, `XLTY` will not be recognized as CLISP. In some cases, DWIM will be able to diagnose this situation as a run-on spelling error, in which case after the atom is split apart, CLISP will be able to perform the indicated transformation.

A number of Lisp functions, such as `EQUAL`, `MEMBER`, `AND`, `OR`, etc., can also be treated as CLISP infix operators. New infix operators can be easily added (see the CLISP Internal Convetions section below). Spelling correction on misspelled infix operators is peformed using `CLISPINFIXSPLST` as a spelling list.

`AND` is higher than `OR`, and both `AND` and `OR` are lower than the other infix operators, so `(X OR Y AND Z)` is the same as `(X OR (Y AND Z))`, and `(X AND Y EQUAL Z)` is the same as `(X AND (Y EQUAL Z))`. All of the infix predicates have lower precedence than Interlisp forms, since it is far more common to apply a predicate to two forms, than to use a Boolean as an argument to a function. Therefore, `(FOO X GT FIE Y)` is translated as `((FOO X) GT (FIE Y))`, rather than as `(FOO (X GT (FIE Y)))`. However, you can easily change this.

:	[CLISP Operator]

X:*N* extracts the *N*th element of the list *X*. `FOO:3` specifies the third element of `FOO`, or `(CADDR FOO)`. If *N* is less than zero, this indicates elements counting from the end of the list; i.e. `FOO:-1` is the last element of `FOO`. `:` operators can be nested, so `FOO:1:2` means the second element of the first element of `FOO`, or `(CADAR FOO)`.

The : operator can also be used for extracting substructures of records (see Chapter 8). Record operations are implemented by replacing expressions of the form `X:FOO` by `(fetch FOO of X)`. Both lower- and uppercase are acceptable.

: is also used to indicate operations in the pattern match facility (see Chapter 12). `X:(&` `'A -- 'B)` translates to (match `X` with `(& 'A -- 'B)`)

. [CLISP Operator]

In combination with :, a period can be used to specify the "data path" for record operations. For example, if `FOO` is a field of the `BAR` record, `X:BAR.FOO` is translated into (fetch (`BAR  FOO`) of `X`). Subrecord fields can be specified with multiple periods: `X:BAR.FOO.BAZ` translates into (fetch (`BAR FOO BAZ`) of `X`).

> Note: If a record contains fields with periods in them, `CLISPIFY` will not translate a record operation into a form using periods to specify the data path. For example, `CLISPIFY` will NOT translate (fetch `A.B` of `X`) into `X:A.B`.

:: [CLISP Operator]

X:*N*, returns the *N*th *tail* of the list *X*. For example, `FOO::3` is (CDDDR `FOO`), and `FOO::-1` is (LAST `FOO`).

← [CLISP Operator]

← is used to indicate assignment. For example, `X←Y` translates to (SETQ `X Y`). If `X` does not have a value, and is not the name of one of the bound variables of the function in which it appears, spelling correction is attempted. However, since this may simply be a case of assigning an initial value to a new free variable, DWIM will always ask for approval before making the correction.

In conjunction with : and ::, ← can also be used to perform a more general type of assignment, involving structure modification. For example, `X:2←Y` means "make the second element of `X` *be* `Y`", in Interlisp terms (RPLACA (CDR `X`) `Y`). Note that the *value* of this operation is the value of RPLACA, which is (CDR `X`), rather than `Y`. Negative numbers can also be used, e.g., `X:-2_Y`, which translates to (RPLACA (NLEFT `X` 2) `Y`).

You can indicate you want /RPLACA and /RPLACD used (undoable version of RPLACA and RPLACD, see Chapter 13), or FRPLACA and FRPLACD (fast versions of RPLACA and RPLACD, see Chapter 3), by means of CLISP declarations. The initial default is to use RPLACA and RPLACD.

← is also used to indicate assignment in record operations (`X:FOO←Y` translates to (replace FOO of X with Y).), and pattern match operations (Chapter 12).

← has different precedence on the left from on the right. On the left,← is a "tight" operator, i.e., high precedence, so that `A+B←C` is the same as `A+(B←C)`. On the right, ← has broader scope so that `A←B+C` is the same as `A←(B+C)`.

On type-in, $←*FORM* (where $ is the escape key) is equivalent to set the "last thing mentioned", i.e., is equivalent to (SET LASTWORD *FORM*) (see Chapter 20). For example,

immediately after examining the value of LONGVARIABLENAME, you could set it by typing $← followed by a form.

Note that an atom of the form X←Y, appearing at the top level of a PROG, will not be recognized as an assignment statement because it will be interpreted as a PROG label by the Interlisp interpreter, and therefore will not cause an error, so DWIM and CLISP will never get to see it. Instead, one must write (X←Y).

<   [CLISP Operator]
>   [CLISP Operator]

Angle brackets are used in CLISP to indicate list construction. The appearance of a "<" corresponds to a "(" and indicates that a list is to be constructed containing all the elements up to the corresponding ">". For example, <A B <C>> translates to (LIST A B (LIST C)). ! can be used to indicate that the next expression is to be inserted in the list as a *segment*, e.g., <A B ! C> translates to (CONS A (CONS B C)) and <! A ! B C> to (APPEND A B (LIST C)). !! is used to indicate that the next expression is to be inserted as a segment, and furthermore, all list structure to its right in the angle brackets is to be physically attached to it, e.g., <!! A B> translates to (NCONC1 A B), and <!!A !B !C> to (NCONC A (APPEND B C)). Not (NCONC (APPEND A B) C), which would have the same value, but would attach C to B, and not attach either to A. Note that <, !, !!, and > need not be separate atoms, for example, <A B ! C> may be written equally well as < A B !C >. Also, arbitrary Interlisp or CLISP forms may be used within angle brackets. For example, one can write <FOO←(FIE X) ! Y> which translates to (CONS (SETQ FOO (FIE X)) Y). CLISPIFY converts expressions in CONS, LIST, APPEND, NCONC, NCONC1, /NCONC, and /NCONC1 into equivalent CLISP expressions using <, >, !, and !!.

> Note: brackets differ from other CLISP operators. For example, <A B 'C> translates to (LIST A B (QUOTE C)) even though following ', all *operators* are ignored for the rest of the identifier. (This is true only if a previous unmatched < has been seen, e.g., (PRINT 'A>B) will print the atom A>B.) Note however that <A B ' C> D> is equivalent to (LIST A B (QUOTE C>) D).

'   [CLISP Operator]

CLISP recognizes ' as a prefix operator. ' means QUOTE when it is the first character in an identifier, and is ignored when it is used in the interior of an identifier. Thus, X = 'Y means (EQ X (QUOTE Y)), but X = CAN'T means (EQ X CAN'T), *not* (EQ X CAN) followed by (QUOTE T). This enables users to have variable and function names with ' in them (so long as the ' is not the first character).

Following ', all operators are ignored for the rest of the identifier, e.g., '*A means (QUOTE *A), and 'X=Y means (QUOTE X=Y), not (EQ (QUOTE X) Y). To write (EQ (QUOTE X) Y), one writes Y='X, or 'X =Y. This is one place where an extra space does make a difference.

On type-in, '$ (escape) is equivalent to (QUOTE *VALUE-OF-LASTWORD*) (see Chapter 19). For example, after calling PRETTYPRINT on LONGFUNCTION, you could move its definition to FOO by typing (MOVD '$ 'FOO).

Note that this is not (MOVD $ 'FOO), which would be equivalent to (MOVD LONGFUNCTION 'FOO), and would (probably) cause a U.B.A. LONGFUNCTION error, nor MOVD($ FOO), which would actually move the definition of $ to FOO, since DWIM and the spelling corrector would never be invoked.

~   [CLISP Operator]

CLISP recognizes ~ as a prefix operator meaning NOT. ~ can negate a form, as in ~(ASSOC X Y), or ~X, or negate an infix operator, e.g., (A ~GT B) is the same as (A LEQ B). Note that ~A = B means (EQ (NOT A) B).

When ~ negates an operator, e.g., ~=, ~LT, the two operators are treated as a single operator whose precedence is that of the second operator. When ~ negates a function, e.g., (~FOO X Y), it negates the whole form, i.e., (~(FOO X Y)).

Order of Precedence of CLISP Operators:

```
'
:
← (left precedence)
– (unary), ~
↑
*, /
+, – (binary)
← (right precedence)
=
```

Interlisp forms

```
LT, GT, EQUAL, MEMBER, etc.
AND
OR
IF, THEN, ELSEIF, ELSE
iterative statement operators
```

Declarations

CLISP declarations are used to affect the choice of Interlisp function used as the translation of a particular operator. For example, A+B can be translated as either (PLUS A B), (FPLUS A B), or (IPLUS A B), depending on the declaration in effect. Similarly X:1←Y can mean (RPLACA X Y), (FRPLACA X Y), or (/RPLACA X Y), and <!! A B> either (NCONC1 A B) or (/NCONC1 A B). Note that the choice of function on all CLISP transformations are affected by the CLISP declaration in

effect, i.e., iterative statements, pattern matches, record operations, as well as infix and prefix operators.

(**CLISPDEC** *DECLST*)   [Function]

Puts into effect the declarations in *DECLST*. CLISPDEC performs spelling corrections on words not recognized as declarations. CLISPDEC is undoable.

You can makes (changes) a global declaration by calling CLISPDEC with *DECLST* a list of declarations, e.g., (CLISPDEC '(FLOATING UNDOABLE)). Changing a global declaration does not affect the speed of subsequent CLISP transformations, since all CLISP transformation are table driven (i.e., property list), and global declarations are accomplished by making the appropriate internal changes to CLISP at the time of the declaration. If a function employs *local* declarations (described below), there will be a slight loss in efficiency owing to the fact that for each CLISP transformation, the declaration list must be searched for possibly relevant declarations.

Declarations are implemented in the order that they are given, so that later declarations override earlier ones. For example, the declaration FAST specifies that FRPLACA, FRPLACD, FMEMB, and FLAST be used in place of RPLACA, RPLACD, MEMB, and LAST; the declaration RPLACA specifies that RPLACA be used. Therefore, the declarations (FAST RPLACA RPLACD) will cause FMEMB, FLAST, RPLACA, and RPLACD to be used.

The initial global declaration is MIXED and STANDARD.

The table below gives the declarations available in CLISP, and the Interlisp functions they indicate:

Declaration:	Interlisp Functions to be used:
MIXED	PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, LESSP, GREATERP
INTEGER or FIXED	IPLUS, IMINUS, IDIFFERENCE, ITIMES, IQUOTIENT, ILESSP, IGREATERP
FLOATING	FPLUS, FMINUS, FDIFFERENCE, FTIMES, FQUOTIENT, LESSP, FGREATERP
FAST	FRPLACA, FRPLACD, FMEMB, FLAST, FASSOC
UNDOABLE	/RPLACA, /RPLACD, /NCONC, /NCONC1, /MAPCONC, /MAPCON
STANDARD	RPLACA, RPLACD, MEMB, LAST, ASSOC, NCONC, NCONC1, MAPCONC, MAPCON
RPLACA, RPLACD, /RPLACA, etc.	corresponding function

You can also make local declarations affecting a selected function or functions by inserting an expression of the form (CLISP: . *DECLARATIONS*) immediately following the

argument list, i.e., as CADDR of the definition. Such local declarations take precedence over global declarations. Declarations affecting selected variables can be indicated by lists, where the first element is the name of a variable, and the rest of the list the declarations for that variable. For example, (CLISP: FLOATING (X INTEGER)) specifies that in this function integer arithmetic be used for computations involving X, and floating arithmetic for all other computations, where "involving" means where the variable itself is an operand. For example, with the declaration (FLOATING (X INTEGER)) in effect, (FOO X)+(FIE X) would translate to FPLUS, i.e., use floating arithmetic, even though X appears somewhere inside of the operands, whereas X+(FIE X) would translate to IPLUS. If there are declarations involving *both* operands, e.g., X+Y, with (X FLOATING) (Y INTEGER), whichever appears first in the declaration list will be used.

You can also make local record declarations by inserting a record declaration, e.g., (RECORD --),(ARRAYRECORD --), etc., in the local declaration list. In addition, a local declaration of the form (RECORDS A B C) is equivalent to having copies of the global declarations A, B, and C in the local declaration. Local record declarations override global record declarations for the function in which they appear. Local declarations can also be used to override the global setting of certain DWIM/CLISP parameters effective only for transformations within that function, by including in the local declaration an expression of the form (*VARIABLE* = *VALUE*), e.g., (PATVARDEFAULT = QUOTE).

The CLISP: expression is converted to a comment of a special form recognized by CLISP. Whenever a CLISP transformation that is affected by declarations is about to be performed in a function, this comment will be searched for a relevant declaration, and if one is found, the corresponding function will be used. Otherwise, if none are found, the global declaration(s) currently in effect will be used.

Local declarations are effective in the order that they are given, so that later declarations can be used to override earlier ones, e.g., (CLISP: FAST RPLACA RPLACD) specifies that FMEMB, FLAST, RPLACA, and RPLACD be used. An exception to this is that declarations for specific variables take precedence of general, function-wide declarations, regardless of the order of appearance, as in (CLISP: (X INTEGER) FLOATING).

CLISPIFY also checks the declarations in effect before selecting an infix operator to ensure that the corresponding CLISP construct would in fact translate back to this form. For example, if a FLOATING declaration is in effect, CLISPIFY will convert (FPLUS X Y) to X+Y, but leave (IPLUS X Y) as is. If (FPLUS X Y) is CLISPIFYed while a FLOATING declaration is under effect, and then the declaration is changed to INTEGER, when X+Y is translated back to Interlisp, it will become (IPLUS X Y).

CLISP Operation

CLISP is a part of the basic Medley system. Without any special preparations, you can include CLISP constructs in programs, or type them in directly for evaluation (in EVAL or APPLY format), then, when the "error" occurs, and DWIM is called, it will destructively transform the CLISP to the equivalent Interlisp expression and evaluate the Interlisp expression. CLISP transformations, like all DWIM

corrections, are undoable. User approval is not requested, and no message is printed. This entire discussion also applies to CLISP transformation initiated by calls to DWIM from `DWIMIFY`.

However, if a CLISP construct contains an error, an appropriate diagnostic is generated, and the form is left unchanged. For example, if you write `(LIST X+Y*)`, the error diagnostic `MISSING OPERAND AT X+Y* IN (LIST X+Y*)` would be generated. Similarly, if you write `(LAST+EL X)`, CLISP knows that `((IPLUS LAST EL) X)` is not a valid Interlisp expression, so the error diagnostic `MISSING OPERATOR IN (LAST+EL X)` is generated. (For example, you might have meant to say `(LAST+EL*X)`.) If `LAST+EL` were the name of a defined function, CLISP would never see this form.

Since the bad CLISP transformation might not be CLISP at all, for example, it might be a misspelling of a user function or variable, DWIM holds all CLISP error messages until after trying other corrections. If one of these succeeds, the CLISP message is discarded. Otherwise, if all fail, the message is printed (but no change is made).   For example, suppose you type `(R/PLACA X Y)`. CLISP generates a diagnostic, since `((IQUOTIENT R PLACA) X Y)` is obviously not right. However, since `R/PLACA` spelling corrects to `/RPLACA`, this diagnostic is never printed.

> Note: CLISP error messages are not printed on type-in. For example, typing `X+*Y` will just produce a `U.B.A. X+*Y` message.

If a CLISP infix construct is well formed from a syntactic standpoint, but one or both of its operands are atomic and not bound, it is possible that either the operand is misspelled, e.g., you wrote `X+YY` for `X+Y`, or that a CLISP transformation operation was not intended at all, but that the entire expression is a misspelling. For the purpose of `DWIMIFY`ing, "not bound" means no top level value, not on list of bound variables built up by `DWIMIFY` during its analysis of the expression, and not on `NOFIXVARSLST`, i.e., not previously seen.

For example, if you have a variable named `LAST-EL`, and write `(LIST LAST-ELL)`. Therefore, CLISP computes, but does not actually perform, the indicated infix transformation. DWIM then continues, and if it is able to make another correction, does so, and ignores the CLISP interpretation. For example, with `LAST-ELL`, the transformation `LAST-ELL -> LAST-EL` would be found.

If no other transformation is found, and DWIM is about to interpret a construct as CLISP for which one of the operands is not bound, DWIM will ask you whether CLISP was intended, in this case by printing `LAST-ELL TREAT AS CLISP ?`.

> Note: If more than one infix operator was involved in the CLISP construct, e.g., `X+Y+Z`, or the operation was an assignment to a variable already noticed, or `TREATASCLISPFLG` is `T` (initially `NIL`), you will simply be informed of the correction, e.g., `X+Y+Z TREATED AS CLISP`. Otherwise, even if DWIM was enabled in `TRUSTING` mode, you will be asked to approve the correction.

The same sort of procedure is followed with 8 and 9 errors. For example, suppose you write `FOO8*X` where `FOO8` is not bound. The CLISP transformation is noted, and DWIM proceeds. It next asks you to approve `FOO8*X -> FOO ( *X`. For example, this would make sense if you have (or plan to define) a function named `*X`. If you refuses, you are asked whether `FOO8*X` is to be treated as CLISP.

Similarly, if `FOO8` were the name of a variable, and you write `FOOO8*X`, you will first be asked to approve `FOOO8*X -> FOOO ( XX`, and if you refuse, then be offered the `FOOO8 -> FOO8` correction. The 8-9 transformation is tried before spelling correction since it is empirically more likely that an unbound atom or undefined function containing an 8 or a 9 is a parenthesis error, rather than a spelling error.

CLISP also contains provision for correcting misspellings of infix operators (other than single characters), `IF` words, and i.s. operators. This is implemented in such a way that the user who does not misspell them is not penalized. For example, if you write `IF N = 0 THEN 1 ELSSE N*(FACT N-1)` CLISP does *not* operate by checking each word to see if it is a misspelling of `IF`, `THEN`, `ELSE`, or `ELSEIF`, since this would seriously degrade CLISP's performance on *all* `IF` statements. Instead, CLISP assumes that all of the `IF` words are spelled correctly, and transforms the expression to `(COND ((ZEROP N) 1 ELSSE N*(FACT N-1)))`. Later, after DWIM cannot find any other interpretation for `ELSSE`, and using the fact that this atom originally appeared in an `IF` statement, DWIM attempts spelling correction, using `(IF THEN ELSE ELSEIF)` for a spelling list. When this is successful, DWIM "fails" all the way back to the original `IF` statement, changes `ELSSE` to `ELSE`, and starts over. Misspellings of `AND`, `OR`, `LT`, `GT`, etc. are handled similarly.

CLISP also contains many Do-What-I-Mean features besides spelling corrections. For example, the form `(LIST +X Y)` would generate a `MISSING OPERATOR` error. However, `(LIST -X Y)` makes sense, if the minus is unary, so DWIM offers this interpretation to you. Another common error, especially for new users, is to write `(LIST X*FOO(Y))` or `(LIST X*FOO Y)`, where `FOO` is the name of a function, instead of `(LIST X*(FOO Y))`. Therefore, whenever an operand that is not bound is also the name of a function (or corrects to one), the above interpretations are offered.

CLISP Translations

The translation of CLISP character operators and the CLISP word `IF` are handled by *replacing* the CLISP expression with the corresponding Interlisp expression, and discarding the original CLISP. This is done because (1) the CLISP expression is easily recomputable (by `CLISPIFY`) and (2) the Interlisp expressions are simple and straightforward. Another reason for discarding the original CLISP is that it may contain errors that were corrected in the course of translation (e.g., `FOO←FOOO:1`, `N*8FOO X`), etc.). If the original CLISP were retained, either you would have to go back and fix these errors by hand, thereby negating the advantage of having DWIM perform these corrections, or else DWIM would have to keep correcting these errors over and over.

Note that `CLISPIFY` is sufficiently fast that it is practical for you to configure your Interlisp system so that all expressions are automatically `CLISPIFY`ed immediately before they are presented to you. For example, you can define an edit macro to use in place of `P` which calls `CLISPIFY` on the current expression before printing it. Similarly, you can inform `PRETTYPRINT` to call `CLISPIFY` on each expression before printing it, etc.

Where (1) or (2) are not the case, e.g., with iterative statements, pattern matches, record expressions, etc. the original CLISP *is* retained (or a slightly modified version thereof), and the translation is stored

elsewhere (by the function `CLISPTRAN`, in the Miscellaneous Functions and Variables), usually in the hash array `CLISPARRAY`. The interpreter automatically checks this array when given a form `CAR` of which is not a function. Similarly, the compiler performs a `GETHASH` when given a form it does not recognize to see if it has a translation, which is then compiled instead of the form. Whenever you *change* a CLISP expresson by editing it, the editor automatically deletes its translation (if one exists), so that the next time it is evaluated or `DWIMIFI`ed, the expression will be retranslated (if the value of `CLISPRETRANFLG` is `T`, `DWIMIFY` will also (re)translate any expressions which have translations stored remotely, see the CLISPIFY section). The function `PPT` and the edit commands `PPT` and `CLISP:` are available for examining translations (see the Miscellaneous Functions and Variables section).

You can also indicate that you want the original CLISP retained by embedding it in an expression of the form (`CLISP . CLISP-EXPRESSION`), e.g., (`CLISP X:5:3`) or (`CLISP <A B C ! D>`). In such cases, the translation will be stored remotely as described above. Furthermore, such expressions will be treated as CLISP even if infix and prefix transformations have been disabled by setting `CLISPFLG` to `NIL` (see the Miscellaneous Functions and Variables section). In other words, you can instruct the system to interpret as CLISP infix or prefix constructs only those expressions that are specifically flagged as such. You can also include CLISP declarations by writing (`CLISP DECLARATIONS . FORM`), e.g., (`CLISP (CLISP: FLOATING) ...`). These declarations will be used in place of any CLISP declarations in the function definition. This feature provides a way of including CLISP declarations in macro definitions.

> Note: CLISP translations can also be used to supply an interpretation for function objects, as well as forms, either for function objects that are used openly, i.e., appearing as `CAR` of form, function objects that are explicitly `APPLY`ed, as with arguments to mapping functions, or function objects contained in function definition cells. In all cases, if `CAR` of the object is not `LAMBDA` or `NLAMBDA`, the interpreter and compiler will check `CLISPARRAY`.

DWIMIFY

`DWIMIFY` is effectively a preprocessor for CLISP. `DWIMIFY` operates by scanning an expression as though it were being interpreted, and for each form that would generate an error, calling DWIM to "fix" it. `DWIMIFY` performs *all* DWIM transformations, not just CLISP transformations, so it does spelling correction, fixes 8-9 errors, handles `F/L`, etc. Thus you will see the same messages, and be asked for approval in the same situations, as you would if the expression were actually run. If DWIM is unable to make a correction, no message is printed, the form is left as it was, and the analysis proceeds.

`DWIMIFY` knows exactly how the interpreter works. It knows the syntax of `PROG`s, `SELECTQ`s, `LAMBDA` expressions, `SETQ`s, et al. It knows how variables are bound, and that the argument of `NLAMBDA`s are not evaluated (you can inform `DWIMIFY` of a function or macro's nonstandard binding or evaluation by giving it a suitable `INFO` property, see below). In the course of its analysis of a particular expression, `DWIMIFY` builds a list of the bound variables from the `LAMBDA` expressions and `PROG`s that it encounters. It uses this list for spelling corrections. `DWIMIFY` also knows not to try to

"correct" variables that are on this list since they would be bound if the expression were actually being run. However, note that DWIMIFY cannot, a priori, know about variables that are used freely but would be bound in a higher function if the expression were evaluated in its normal context. Therefore, DWIMIFY will try to "correct" these variables. Similarly, DWIMIFY will attempt to correct forms for which CAR is undefined, even when the form is not in error from your standpoint, but the corresponding function has simply not yet been defined.

>   Note: DWIMIFY rebinds FIXSPELLDEFAULT to N, so that if you are not at the terminal
>   when DWIMIFYing (or compiling), spelling corrections will not be performed.

DWIMIFY will also inform you when it encounters an expression with too *many* arguments (unless DWIMCHECK#ARGSFLG = NIL), because such an occurrence, although does not cause an error in the Interlisp interpreter, nevertheless is frequently symptomatic of a parenthesis error. For example, if you wrote (CONS (QUOTE FOO X)) instead of (CONS (QUOTE FOO) X), DWIMIFY will print:

```
POSSIBLE PARENTHESIS ERROR IN
(QUOTE FOO X)
TOO MANY ARGUMENTS (MORE THAN 1)
```

DWIMIFY will also check to see if a PROG label contains a clisp character (unless DWIMCHECKPROGLABELSFLG = NIL, or the label is a member of NOFIXVARSLST), and if so, will alert you by printing the message SUSPICIOUS PROG LABEL, followed by the label. The PROG label will *not* be treated as CLISP.

Note that in most cases, an attempt to transform a form that is already as you intended will have no effect (because there will be nothing to which that form could reasonably be transformed). However, in order to avoid needless calls to DWIM or to avoid possible confusion, you can inform DWIMIFY *not* to attempt corrections or transformations on certain functions or variables by adding them to the list NOFIXFNSLST or NOFIXVARSLST respectively. Note that you could achieve the same effect by simply setting the corresponding variables, and giving the functions dummy definitions.

DWIMIFY will never attempt corrections on global variables, i.e., variables that are a member of the list GLOBALVARS, or have the property GLOBALVAR with value T, on their property list. Similarly, DWIMIFY will not attempt to correct variables declared to be SPECVARS in block declarations or via DECLARE expressions in the function body. You can also declare variables that are simply used freely in a function by using the USEDFREE declaration.

DWIMIFY and DWIMIFYFNS (used to DWIMIFY several functions) maintain two internal lists of those functions and variables for which corrections were unsuccessfully attempted. These lists are initialized to the values of NOFIXFNSLST and NOFIXVARSLST. Once an attempt is made to fix a particular function or variable, and the attempt fails, the function or variable is added to the corresponding list, so that on subsequent occurrences (within this call to DWIMIFY or DWIMIFYFNS), no attempt at correction is made. For example, if FOO calls FIE several times, and FIE is undefined at the time FOO is DWIMIFYed, DWIMIFY will not bother with FIE after the first occurrence. In other words, once DWIMIFY "notices" a function or variable, it no longer attempts to correct it. DWIMIFY and DWIMIFYFNS also "notice" free variables that are set in the expression being processed.

Moreover, once DWIMIFY "notices" such functions or variables, it subsequently treats them the same as though they were actually defined or set.

Note that these internal lists are local to each call to DWIMIFY and DWIMIFYFNS, so that if a function containing FOOO, a misspelled call to FOO, is DWIMIFYed before FOO is defined or mentioned, if the function is DWIMIFYed again after FOO has been defined, the correction will be made.

You can undo selected transformations performed by DWIMIFY, as described in Chapter 13.

(**DWIMIFY** *X QUIETFLG L*) [Function]

> Performs all DWIM and CLISP corrections and transformations on *X* that would be performed if *X* were run, and prints the result unless *QUIETFLG* = T.

> If *X* is an atom and *L* is NIL, *X* is treated as the name of a function, and its entire definition is DWIMIFYed. If *X* is a list or *L* is not NIL, *X* is the expression to be DWIMIFYed. If *L* is not NIL, it is the edit push-down list leading to *X*, and is used for determining context, i.e., what bound variables would be in effect when *X* was evaluated, whether *X* is a form or sequence of forms, e.g., a COND clause, etc.

> If *X* is an iterative statement and *L* is NIL, DWIMIFY will also print the translation, i.e., what is stored in the hash array.

(**DWIMIFYFNS** *FN_1* ... *FN_N*) [NLambda NoSpread Function]

> DWIMIFYs each of the functions given. If only one argument is given, it is evalued. If its value is a list, the functions on this list are DWIMIFYed. If only one argument is given, it is atomic, its value is not a list, and it is the name of a known file, DWIMIFYFNS will operate on (FILEFNSLST *FN_1*), e.g. (DWIMIFYFNS FOO.LSP) will DWIMIFY every function in the file FOO.LSP.

> Every 30 seconds, DWIMIFYFNS prints the name of the function it is processing, a la PRETTYPRINT.

> Value is a list of the functions DWIMIFYed.

DWIMINMACROSFLG [Variable]

> Controls how DWIMIFY treats the arguments in a "call" to a macro, i.e., where the CAR of the form is undefined, but has a macro definition. If DWIMINMACROSFLG is T, then macros are treated as LAMBDA functions, i.e., the arguments are assumed to be evaluated, which means that DWIMIFY will descend into the argument list. If DWIMINMACROSFLG is NIL, macros are treated as NLAMBDA functions. DWIMINMACROSFLG is initially T.

INFO [Property Name]

> Used to inform DWIMIFY of nonstandard behavior of particular forms with respect to evaluation, binding of arguments, etc. The INFO property of a symbol is a single atom or list of atoms chosen from among the following:

EVAL   Informs DWIMIFY (and CLISP and Masterscope) that an nlambda function *does* evaluate its arguments. Can also be placed on a macro name to override the behavior of DWIMINMACROSFLG = NIL.

NOEVAL   Informs DWIMIFY that a macro does *not* evaluate all of its arguments, even when DWIMINMACROSFLG = T.

BINDS   Placed on the INFO property of a function or the CAR of a special form to inform DWIMIFY that the function or form binds variables. In this case, DWIMIFY assumes that CADR of the form is the variable list, i.e., a list of symbols, or lists of the form (*VAL VALUE*). LAMBDA, NLAMBDA, PROG, and RESETVARS are handled in this fashion.

LABELS   Informs CLISPIFY that the form interprets top-level symbols as labels, so that CLISPIFY will never introduce an atom (by packing) at the top level of the expression. PROG is handled in this fashion.

NOFIXFNSLST   [Variable]

List of functions that DWIMIFY will not try to correct.

NOFIXVARSLST   [Variable]

List of variables that DWIMIFY will not try to correct.

NOSPELLFLG   [Variable]

If T, DWIMIFY will not perform any spelling corrections. Initially NIL. NOSPELLFLG is reset to T when compiling functions whose definitions are obtained from a file, as opposed to being in core.

CLISPHELPFLG   [Variable]

If NIL, DWIMIFY will not ask you for approval of any CLISP transformations. Instead, in those situations where approval would be required, the effect is the same as though you had been asked and said NO. Initially T.

DWIMIFYCOMPFLG   [Variable]

If T, DWIMIFY is called before compiling an expression. Initially NIL.

DWIMCHECK#ARGSFLG   [Variable]

If T, causes DWIMIFY to check for too many arguments in a form. Initially T.

DWIMCHECKPROGLABELSFLG   [Variable]

If T, causes DWIMIFY to check whether a PROG label contains a CLISP character. Initially T.

DWIMESSGAG   [Variable]

If `T`, suppresses all `DWIMIFY` error messages. Initially `NIL`.

CLISPRETRANFLG   [Variable]

If `T`, informs `DWIMIFY` to (re)translate all expressions which have remote translations in the CLISP hash array. Initially `NIL`.

CLISPIFY

`CLISPIFY` converts Interlisp expressions to CLISP. Note that the expression given to `CLISPIFY` need *not* have originally been input as CLISP, i.e., `CLISPIFY` can be used on functions that were written before CLISP was even implemented. `CLISPIFY` is cognizant of declaration rules as well as all of the precedence rules. For example, `CLISPIFY` will convert `(IPLUS A (ITIMES B C))` into `A+B*C`, but `(ITIMES A (IPLUS B C))` into `A*(B+C)`. `CLISPIFY` handles such cases by first `DWIMIFY`ing the expression. `CLISPIFY` also knows how to handle expressions consisting of a mixture of Interlisp and CLISP, e.g., `(IPLUS A B*C)` is converted to `A+B*C`, but `(ITIMES A B+C)` to `(A*(B+C))`. `CLISPIFY` converts calls to the six basic mapping functions, `MAP`, `MAPC`, `MAPCAR`, `MAPLIST`, `MAPCONC`, and `MAPCON`, into equivalent iterative statements. It also converts certain easily recognizable internal `PROG` loops to the corresponding iterative statements. `CLISPIFY` can convert all iterative statements input in CLISP back to CLISP, regardless of how complicated the translation was, because the original CLISP is saved.

`CLISPIFY` is not destructive to the original Interlisp expression, i.e., `CLISPIFY` produces a new expression without changing the original. The new expression may however contain some "pieces" of the original, since `CLISPIFY` attempts to minimize the number of `CONS`es by not copying structure whenever possible.

`CLISPIFY` will not convert expressions appearing as arguments to `NLAMBDA` functions, except for those functions whose `INFO` property is or contains the atom `EVAL`. `CLISPIFY` also contains built in information enabling it to process special forms such as `PROG`, `SELECTQ`, etc. If the `INFO` property is or contains the atom `LABELS`, `CLISPIFY` will never create an atom (by packing) at the top level of the expression. `PROG` is handled in this fashion.

> Note: Disabling a CLISP operator with `CLDISABLE` (see the Miscellaneous Functions and Variables section) will also disable the corresponding `CLISPIFY` transformation. Thus, if ← is "turned off", A←B will not transform to `(SETQ A B)`, nor vice versa.

(**CLISPIFY** *X EDITCHAIN*)   [Function]

Clispifies *X*. If *X* is an atom and *EDITCHAIN* is `NIL`, *X* is treated as the name of a function, and its definition (or `EXPR` property) is clispified. After `CLISPIFY` has finished, *X* is redefined (using `/PUTD`) with its new CLISP definition. The value of `CLISPIFY` is *X*. If *X*

is atomic and not the name of a function, spelling correction is attempted. If this fails, an error is generated.

If `X` is a list, or *EDITCHAIN* is not `NIL`, `X` itself is the expression to be clispified. If *EDITCHAIN* is not `NIL`, it is the edit push-down list leading to `X` and is used to determine context as with `DWIMIFY`, as well as to obtain the local declarations, if any. The value of `CLISPIFY` is the clispified version of `X`.

(**CLISPIFYFNS** FN_1 ... FN_N)   [NLambda NoSpread Function]

Like `DWIMIFYFNS` except calls `CLISPIFY` instead of `DWIMIFY`.

CL:FLG   [Variable]

Affects `CLISPIFY`'s handling of forms beginning with `CAR`, `CDR`, ... `CDDDDR`, as well as pattern match and record expressions. If `CL:FLG` is `NIL`, these are not transformed into the equivalent `:` expressions. This will prevent `CLISPIFY` from constructing any expression employing a `:` infix operator, e.g., (`CADR X`) will not be transformed to `X:2`. If `CL:FLG` is `T`, `CLISPIFY` will convert to `:` notation only when the argument is atomic or a simple list (a function name and one atomic argument). If `CL:FLG` is `ALL`, `CLISPIFY` will convert to `:` expressions whenever possible.

`CL:FLG` is initially `T`.

CLREMPARSFLG   [Variable]

If `T`, `CLISPIFY` will remove parentheses in certain cases from simple forms, where "simple" means a function name and one or two atomic arguments. For example, (`COND ((ATOM X) --)`) will `CLISPIFY` to (`IF ATOM X THEN --`). However, if `CLREMPARSFLG` is set to `NIL`, `CLISPIFY` will produce (`IF (ATOM X) THEN --`). Regardless of the flag setting, the expression can be input in either form.

`CLREMPARSFLG` is initially `NIL`.

CLISPIFYPACKFLG   [Variable]

`CLISPIFYPACKFLG` affects the treatment of infix operators with atomic operands. If `CLISPIFYPACKFLG` is `T`, `CLISPIFY` will pack these into single atoms, e.g., (`IPLUS A (ITIMES B C)`) becomes `A+B*C`. If `CLISPIFYPACKFLG` is `NIL`, no packing is done, e.g., the above becomes `A + B * C`.

`CLISPIFYPACKFLG` is initially `T`.

CLISPIFYUSERFN   [Variable]

If `T`, causes the function `CLISPIFYUSERFN`, which should be a function of one argument, to be called on each form (list) not otherwise recognized by `CLISPIFY`. If a non-`NIL` value is returned, it is treated as the clispified form. Initially `NIL`

Note that `CLISPIFYUSERFN` must be both set and defined to use this feature.

FUNNYATOMLST [Variable]

Suppose you have variables named A, B, and A*B. If CLISPIFY were to convert (ITIMES A B) to A*B, A*B would not translate back correctly to (ITIMES A B), since it would be the name of a variable, and therefore would not cause an error. You can prevent this from happening by adding A*B to the list FUNNYATOMLST. Then, (ITIMES A B) would CLISPIFY to A * B.

Note that A*B's appearance on FUNNYATOMLST would *not* enable DWIM and CLISP to decode A*B+C as (IPLUS A*B C); FUNNYATOMLST is used only by CLISPIFY. Thus, if an identifier contains a CLISP character, it should always be separated (with spaces) from other operators. For example, if X* is a variable, you should write (SETQ X* *FORM*) in CLISP as X* ←*FORM*, not X*←*FORM*. In general, it is best to avoid use of identifiers containing CLISP character operators as much as possible.

Miscellaneous Functions and Variables

CLISPFLG [Variable]

If CLISPFLG = NIL, disables all CLISP infix or prefix transformations (but does not affect IF/THEN/ELSE statements, or iterative statements).

If CLISPFLG = TYPE-IN, CLISP transformations are performed only on expressions that are typed in for evaluation, i.e., not on user programs.

If CLISPFLG = T, CLISP transformations are performed on all expressions.

The initial value for CLISPFLG is T. CLISPIFYing anything will cause CLISPFLG to be set to T.

CLISPCHARS [Variable]

A list of the operators that can appear in the interior of an atom. Currently (+ - * / ↑ ~ ' = ← : < > +- ~= @ !).

CLISPCHARRAY [Variable]

A bit table of the characters on CLISPCHARS used for calls to STRPOSL (Chapter 4). CLISPCHARRAY is initialized by performing (SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS)).

CLISPINFIXSPLST [Variable]

A list of infix operators used for spelling correction.

CLISPARRAY [Variable]

Hash array used for storing CLISP translations. CLISPARRAY is checked by FAULTEVAL and FAULTAPPLY on erroneous forms before calling DWIM, and by the compiler.

(**CLEARCLISPARRAY** *NAME --*)   [Function]

> Macro and CLISP expansions are cached in CLISPARRAY, the systems CLISP hash array. When anything changes that would invalidate an expansion, it needs to be removed from the cache. CLEARCLISPARRAY does this for you. The system does this automatically whenever you define redefine a CLISP or macro form. If you have changed something that a CLISP word or a macro depends on the system will not be able to detect this, so you will have to invalidate the cahce by calling CLEARCLISPARRAY. You can clear the whole cache by calling (CLRHASH CLISPARRAY).

(**CLISPTRAN** *X TRAN*)   [Function]

> Gives *X* the translation *TRAN* by storing (key *X*, value *TRAN*) in the hash array CLISPARRAY. CLISPTRAN is called for all CLISP translations, via a non-linked, external function call, so it can be advised.

(**CLISPDEC** *DECLST*)   [Function]

> Puts into effect the declarations in *DECLST*. CLISPDEC performs spelling corrections on words not recognized as declarations. CLISPDEC is undoable.

(**CLDISABLE** *OP*)   [Function]

> Disables the CLISP operator *OP*. For example, (CLDISABLE '-) makes – be just another character. CLDISABLE can be used on all CLISP operators, e.g., infix operators, prefix operators, iterative statement operators, etc. CLDISABLE is undoable.

> > Note: Simply removing a character operator from CLISPCHARS will prevent it from being treated as a CLISP operator when it appears as part of an atom, but it will continue to be an operator when it appears as a separate atom, e.g. (FOO + X) vs FOO+X.

CLISPIFTRANFLG   [Variable]

> Affects handling of translations of IF-THEN-ELSE statements (see Chapter 9). If T, the translations are stored elsewhere, and the (modified) CLISP retained. If NIL, the corresponding COND expression replaces the CLISP. Initially T.

CLISPIFYPRETTYFLG   [Variable]

> If non-NIL, causes PRETTYPRINT (and therefore PP and MAKEFILE) to CLISPIFY selected function definitions before printing them according to the following interpretations of CLISPIFYPRETTYFLG:

> > | | |
> > |---|---|
> > | **ALL** | Clispify all functions. |
> > | **T** or **EXPRS** | Clispify all functions currently defined as EXPRs. |
> > | **CHANGES** | Clispify all functions marked as having been changed. |
> > | a list | Clispify all functions in that list. |

CLISPIFYPRETTYFLG is (temporarily) reset to T when MAKEFILE is called with the option CLISPIFY, and reset to CHANGES when the file being dumped has the property FILETYPE value CLISP. CLISPIFYPRETTYFLG is initially NIL.

> Note: If CLISPIFYPRETTYFLG is non-NIL, and the only transformation performed by DWIM are well formed CLISP transformations, i.e., no spelling corrections, the function will *not* be marked as changed, since it would only have to be re-clispified and re-prettyprinted when the file was written out.

(**PPT** *X*)   [NLambda NoSpread Function]

Both a function and an edit macro for prettyprinting translations. It performs a PP after first resetting PRETTYTRANFLG to T, thereby causing any translations to be printed instead of the corresponding CLISP.

CLISP: [Editor Command]

Edit macro that obtains the translation of the correct expression, if any, from CLISPARRAY, and calls EDITE on it.

CL   [Editor Command]

Edit macro. Replaces current expression with CLISPIFYed current expression. Current expression can be an element or tail.

DW   [Editor Command]

Edit macro. DWIMIFYs current expression, which can be an element (atom or list) or tail.

Both CL and DW can be called when the current expression is either an element or a tail and will work properly. Both consult the declarations in the function being edited, if any, and both are undoable.

(**LOWERCASE** *FLG*)   [Function]

If *FLG* = T, LOWERCASE makes the necessary internal modifications so that CLISPIFY will use lower case versions of AND, OR, IF, THEN, ELSE, ELSEIF, and all i.s. operators. This produces more readable output. Note that you can always type in *either* upper or lower case (or a combination), regardless of the action of LOWERCASE. If *FLG* = NIL, CLISPIFY will use uppercase versions of AND, OR, et al. The value of LOWERCASE is its previous "setting". LOWERCASE is undoable. The initial setting for LOWERCASE is T.

CLISP Internal Conventions

CLISP is almost entirely table driven by the property lists of the corresponding infix or prefix operators. For example, much of the information used for translating the + infix operator is stored on the property list of the symbol "+". Thus it is relatively easy to add new infix or prefix operators or change old ones, simply by adding or changing selected property values. (There *is* some built in

information for handling minus, :, ', and ~, i.e., you could not yourself add such "special" operators, although you can disable or redefine them.)

Global declarations operate by changing the LISPFN and CLISPINFIX properties of the appropriate operators.

CLISPTYPE [Property Name]

> The property value of the property CLISPTYPE is the precedence number of the operator: higher values have higher precedence, i.e., are tighter. Note that the actual value is unimportant, only the value relative to other operators. For example, CLISPTYPE for :, ↑, and * are 14, 6, and 4 respectively. Operators with the same precedence group left to right, e.g., / also has precedence 4, so A/B*C is (A/B)*C.

> An operator can have a different left and right precedence by making the value of CLISPTYPE be a dotted pair of two numbers, e.g., CLISPTYPE of ← is (8 . -12). In this case, CAR is the left precedence, and CDR the right, i.e., CAR is used when comparing with operators on the *left*, and CDR with operators on the *right*. For example, A*B←C+D is parsed as A*(B←(C+D)) because the left precedence of ← is 8, which is higher than that of *, which is 4. The right precedence of ← is -12, which is lower than that of +, which is 2.

> If the CLISPTYPE property for any operator is removed, the corresponding CLISP transformation is disabled, as well as the inverse CLISPIFY transformation.

UNARYOP [Property Name]

> The value of property UNARYOP must be T for unary operators or brackets. The operand is always on the right, i.e., unary operators or brackets are always prefix operators.

BROADSCOPE [Property Name]

> The value of property BROADSCOPE is T if the operator has lower precedence than Interlisp forms, e.g., LT, EQUAL, AND, etc. For example, (FOO X AND Y) parses as ((FOO X) AND Y). If the BROADSCOPE property were removed from the property list of AND, (FOO X AND Y) would parse as (FOO (X AND Y)).

LISPFN [Property Name]

> The value of the property LISPFN is the name of the function to which the infix operator translates. For example, the value of LISPFN for ↑ is EXPT, for ' QUOTE, etc. If the value of the property LISPFN is NIL, the infix operator itself is also the function, e.g., AND, OR, EQUAL.

SETFN [Property Name]

> If FOO has a SETFN property FIE, then (FOO --)←X translates to (FIE -- X). For example, if you make ELT be an infix operator, e.g. #, by putting appropriate CLISPTYPE and LISPFN properties on the property list of # then you can also make # followed by ← translate to SETA, e.g., X#N←Y to (SETA X N Y), by putting SETA on the property list of

ELT under the property SETFN. Putting the list (ELT) on the property list of SETA under property SETFN will enable SETA forms to CLISPIFY back to ELT's.

CLISPINFIX   [Property Name]

The value of this property is the CLISP infix to be used in CLISPIFYing. This property is stored on the property list of the corresponding Interlisp function, e.g., the value of property CLISPINFIX for EXPT is ↑, for QUOTE is ' etc.

CLISPWORD   [Property Name]

Appears on the property list of clisp operators which can appear as CAR of a form, such as FETCH, REPLACE, IF, iterative statement operators, etc. Value of property is of the form (*KEYWORD* . *NAME*), where *NAME* is the lowercase version of the operator, and *KEYWORD* is its type, e.g. FORWORD, IFWORD, RECORDWORD, etc.

KEYWORD can also be the name of a function. When the atom appears as CAR of a form, the function is applied to the form and the result taken as the correct form. In this case, the function should either physically change the form, or call CLISPTRAN to store the translation.

As an example, to make & be an infix character operator meaning OR, you could do the following:

```
←(PUTPROP '& 'CLISPTYPE (GETPROP 'OR 'CLISPTYPE))
←(PUTPROP '& 'LISPFN 'OR)
←(PUTPROP '& 'BROADSCOPE T)
←(PUTPROP 'OR 'CLISPINFIX '&)
←(SETQ CLISPCHARS (CONS '& CLISPCHARS))
←(SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS))
```

21. PERFORMANCE ISSUES

This chapter describes a number of areas that often contribute to performance problems in Medley programs. Many performance problems can be improved by optimizing the use of storage, since allocating and reclaiming large amounts of storage is expensive. Another tactic that can sometimes yield performance improvements is to change the use of variable bindings on the stack to reduce variable lookup time. There are a number of tools that can be used to determine which parts of a computation cause performance bottlenecks.

Storage Allocation and Garbage Collection

As an Medley application program runs, it creates data structures (allocated out of free storage space), manipulates them, and then discards them. If there were no way to reclaim this space, over time the Medley memory would fill up, and the computation would come to a halt. Actually, long before this could happen the system would probably become intolerably slow, due to "data fragmentation," which occurs when the data currently in use are spread over many virtual memory pages, so that most of the computer time must be spent swapping disk pages into physical memory. The problem of fragmentation will occur in any situation where the virtual memory is significantly larger than the real physical memory. To reduce swapping, you want to keep the "working set" (the set of pages containing actively referenced data) as small as possible.

You can write programs that don't generate much "garbage" data, or which recycle data, but such programs tend to be complex and hard to debug. Spending effort writing such programs defeats the whole point of using a system with automatic storage allocation. An important part of any Lisp implementation is the "garbage collector" that finds discarded data and reclaims its space.

There are several well-known approaches to garbage collection. One method is the traditional mark-and-sweep, which identifies "garbage" data by marking all accessible data structures, and then sweeping through the data spaces to find all unmarked objects (i.e., not referenced by any other object).   This method is guaranteed to reclaim all garbage, but it takes time proportional to the number of allocated objects, which may be very large. Also, the time that a mark-and-sweep garbage collection takes is independent of the amount of garbage collected; it is possible to sweep through the whole virtual memory, and only recover a small amount of garbage.

For interactive applications, it is not acceptable to have long interruptions in a computation for to garbage collect. Medley solves this problem by using a reference-counting garbage collector. With this scheme, there is a table containing counts of how many times each object is referenced. This table is updated as pointers are created and discarded, incurring a small overhead distributed over the computation as a whole. (Note: References from the stack are not counted, but are handled separately at "sweep" time; thus the vast majority of data manipulations do not cause updates to this table.) At opportune moments, the garbage collector scans this table, and reclaims all objects that are no longer accessible (have a reference count of zero). The pause while objects are reclaimed is only the time for scanning the reference count tables (small) plus time proportional to the amount of garbage that has to

be collected (typically less than a second). "Opportune" times occur when a certain number of cells have been allocated or when the system has been waiting for you to type something for long enough. The frequency of garbage collection is controlled by the functions and variables described below. For the best system performance, it is desirable to adjust these parameters for frequent, short garbage collections, which will not interrupt interactive applications for very long, and which will have the added benefit of reducing data fragmentation, keeping the working set small.

One problem with the Medley garbage collector is that not all garbage is guaranteed to be collected. Circular data structures, which point to themselves directly or indirectly, are never reclaimed, since their reference counts are always at least one. With time, this unreclaimable garbage may increase the working set to unacceptable levels. Some users have worked with the same Medley virtual memory for a very long time, but it is a good idea to occasionally save all of your functions in files, reinitialize Medley, and rebuild your system. Many users end their working day by issuing a command to rebuild their system and then leaving the machine to perform this task in their absence. If the system seems to be spending too much time swapping (an indication of fragmented working set), this procedure is definitely recommended.

Another limitation of the reference-counting garbage collector is that the table in which reference counts are maintained is of fixed size. For typical Lisp objects that are pointed to from exactly one place (e.g., the individual conses in a list), no burden is placed on this table, since objects whose reference count is 1 are not explicitly represented in the table. However, large, "rich" data structures, with many interconnections, backward links, cross references, etc, can contribute many entries to the reference count table. For example, if you created a data structure that functioned as a doubly-linked list, such a structure would contribute an entry (reference count 2) for each element.

When the reference count table fills up, the garbage collector can no longer maintain consistent reference counts, so it stops doing so altogether. At this point, a window appears on the screen with the following message, and the debugger is entered:

```
Internal garbage collector tables have overflowed, due
to too many pointers with reference count greater than 1.
*** The garbage collector is now disabled. ***
Save your work and reload as soon as possible.
```

[This message is slightly misleading, in that it should say "count not equal to 1". In the current implementation, the garbage collection of a large pointer array whose elements are not otherwise pointed to can place a special burden on the table, as each element's reference count simultaneously drops to zero and is thus added to the reference count table for the short period before the element is itself reclaimed.]

If you exit the debugger window (e.g., with the RETURN command), your computation can proceed; however, the garbage collector is no longer operating. Thus, your virtual memory will become cluttered with objects no longer accessible, and if you continue for long enough in the same virtual memory image you will eventually fill up the virtual memory backing store and grind to a halt.

Garbage collection in Medley is controlled by the following functions and variables:

(**RECLAIM**)   [Function]

Initiates a garbage collection. Returns 0.

(**RECLAIMMIN** *N*)   [Function]

Sets the frequency of garbage collection. Interlisp keeps track of the number of cells of any type that have been allocated; when it reaches a given number, a garbage collection occurs. If *N* is non-NIL, this number is set to *N*. Returns the current setting of the number.

RECLAIMWAIT   [Variable]

Medley will invoke a RECLAIM if the system is idle and waiting for your input for RECLAIMWAIT seconds (currently set for 4 seconds).

(**GCGAG** *MESSAGE*)   [Function]

Sets the behavior that occurs while a garbage collection is taking place. If *MESSAGE* is non-NIL, the cursor is complemented during a RECLAIM; if *MESSAGE* = NIL, nothing happens. The value of GCGAG is its previous setting.

(**GCTRP**)   [Function]

Returns the number of cells until the next garbage collection, according to the RECLAIMMIN number.

The amount of storage allocated to different data types, how much of that storage is in use, and the amount of data fragmentation can be determined using the following function:

(**STORAGE** *TYPES PAGETHRESHOLD*)   [Function]

STORAGE prints out a summary, for each data type, of the amount of space allocated to the data type, and how much of that space is currently in use. If *TYPES* is non-NIL, STORAGE only lists statistics for the specified types. *TYPES* can be a symbol or a list of types. If *PAGETHRESHOLD* is non-NIL, then STORAGE only lists statistics for types that have at least *PAGETHRESHOLD* pages allocated to them.

STORAGE prints out a table with the column headings Type, Assigned, Free Items, In use, and Total alloc. Type is the name of the data type. Assigned is how much of your virtual memory is set aside for items of this type. Currently, memory is allocated in quanta of two pages (1024 bytes). The numbers under Assigned show the number of pages and the total number of items that fit on those pages. Free Items shows how many items are available to be allocated (using the create construct, Chapter 8); these constitute the "free list" for that data type. In use shows how many items of this type are currently in use, i.e., have pointers to them and hence have not been garbage collected. If this number is higher than your program seems to warrant, you may want to look for storage leaks. The sum of Free Items and In use is always the same

as the total `Assigned` items. `Total alloc` is the total number of items of this type that have ever been allocated (see `BOXCOUNT`, in the Performance Measuring section below).

> Note: The information about the number of items of type `LISTP` is only approximate, because list cells are allocated in a special way that precludes easy computation of the number of items per page.

> Note: When a data type is redeclared, the data type name is reassigned. Pages which were assigned to instances of the old data type are labeled `**DEALLOC**`.

At the end of the table printout, `STORAGE` prints a "Data Spaces Summary" listing the number of pages allocated to the major data areas in the virtual address space: the space for fixed-length items (including datatypes), the space for variable-length items, and the space for symbols. Variable-length data types such as arrays have fixed-length "headers," which is why they also appear in the printout of fixed-length data types. Thus, the line printed for the `BITMAP` data type says how many bitmaps have been allocated, but the "assigned pages" column counts only the headers, not the space used by the variable-length part of the bitmap. This summary also lists "Remaining Pages" in relation to the largest possible virtual memory, not the size of the virtual memory backing file in use. This file may fill up, causing a `STORAGE FULL` error, long before the "Remaining Pages" numbers reach zero.

`STORAGE` also prints out information about the sizes of the entries on the variable-length data free list. The block sizes are broken down by the value of the variable `STORAGE.ARRAYSIZES`, initially `(4 16 64 256 1024 4096 16384 NIL)`, which yields a printout of the form:

```
        variable-datum free list:
        le 4        26 items;    104 cells.
        le 16       72 items;    783 cells.
        le 64       36 items;    964 cells.
        le 256      28 items;   3155 cells.
        le 1024      3 items;   1175 cells.
        le 4096      5 items;   8303 cells.
        le 16384     3 items;  17067 cells.
        others       1 items;  17559 cells.
```

This information can be useful in determining if the variable-length data space is fragmented. If most of the free space is composed of small items, then the allocator may not be able to find room for large items, and will extend the variable datum space. If this is extended too much, this could cause an `ARRAYS FULL` error, even if there is a lot of space left in little chunks.

(**STORAGE.LEFT**) [Function]

Provides a programmatic way of determining how much storage is left in the major data areas in the virtual address space. Returns a list of the form (*MDSFREE MDSFRAC 8MBFRAC ATOMFREE ATOMFRAC*), where the elements are interpreted as follows:

> *MDSFREE* The number of free pages left in the main data space (which includes both fixed-length and variable-length data types).

MDSFRAC	The fraction of the total possible main data space that is free.
8MBFRAC	The fraction of the total main data space that is free, relative to eight megabytes.
	This number is useful when using Medley on some early computers where the hardware limits the address space to eight megabytes. The function 32MBADDRESSABLE returns non-NIL if the currently running Medley system can use the full 32 megabyte address space.
ATOMFREE	The number of free pages left in the symbol space.
ATOMFRAC	The fraction of the total symbol space that is free.

Note: Another important space resource is the amount of the virtual memory backing file in use (see VMEMSIZE, Chapter 12). The system will crash if the virtual memory file is full, even if the address space is not exhausted.

Variable Bindings

Different implementations of Lisp use different methods of accessing free variables. The binding of variables occurs when a function or a PROG is entered. For example, if the function FOO has the definition (LAMBDA (A B) *BODY*), the variables A and B are bound so that any reference to A or B from *BODY* or any function called from *BODY* will refer to the arguments to the function FOO and not to the value of A or B from a higher level function. All variable names (symbols) have a top level value cell which is used if the variable has not been bound in any function. In discussions of variable access, it is useful to distinguish between three types of variable access: local, special and global. Local variable access is the use of a variable that is bound within the function from which it is used. Special variable access is the use of a variable that is bound by another function. Global variable access is the use of a variable that has not been bound in any function. We will often refer to a variable all of whose accesses are local as a "local variable." Similarly, a variable all of whose accesses are global we call a "global variable."

In a "deep" bound system, a variable is bound by saving on the stack the variable's name together with a value cell which contains that variable's new value. When a variable is accessed, its value is found by searching the stack for the most recent binding (occurrence) and retrieving the value stored there. If the variable is not found on the stack, the variable's top level value cell is used.

In a "shallow" bound system, a variable is bound by saving on the stack the variable name and the variable's old value and putting the new value in the variable's top level value cell. When a variable is accessed, its value is always found in its top level value cell.

The deep binding scheme has one disadvantage: the amount of cpu time required to fetch the value of a variable depends on the stack distance between its use and its binding. The compiler can determine

local variable accesses and compiles them as fetches directly from the stack. Thus this computation cost only arises in the use of variable not bound in the local frame ("free" variables). The process of finding the value of a free variable is called free variable lookup.

In a shallow bound system, the amount of cpu time required to fetch the value of a variable is constant regardless of whether the variable is local, special or global. The disadvantages of this scheme are that the actual binding of a variable takes longer (thus slowing down function call), the cells that contain the current in use values are spread throughout the space of all symbol value cells (thus increasing the working set size of functions) and context switching between processes requires unwinding and rewinding the stack (thus effectively prohibiting the use of context switching for many applications).

Medley uses deep binding, because of the working set considerations and the speed of context switching. The free variable lookup routine is microcoded, thus greatly reducing the search time. In benchmarks, the largest percentage of free variable lookup time was 20 percent of the total ellapsed time; the normal time was between 5 and 10 percent.

Because of the deep binding, you can sometimes significantly improve performance by declaring global variables. If a variable is declared global, the compiler will compile an access to that variable as a retrieval of its top level value, completely bypassing a stack search. This should be done only for variables that are never bound in functions, such as global databases and flags.

Global variable declarations should be done using the GLOBALVARS file manager command (Chapter 17). Its form is (GLOBALVARS VAR_1 ... VAR_N).

Another way of improving performance is to declare variables as local within a function. Normally, all variables bound within a function have their names put on the stack, and these names are scanned during free variable lookup. If a variable is declared to be local within a function, its name is not put on the stack, so it is not scanned during free variable lookup, which may increase the speed of lookups. The compiler can also make some other optimizations if a variable is known to be local to a function.

A variable may be declared as local within a function by including the form (DECLARE (LOCALVARS VAR_1 ... VAR_N)) following the argument list in the definition of the function. Local variable declarations only effect the compilation of a function. Interpreted functions put all of their variable names on the stack, regardless of any declarations.

Performance Measuring

This section describes functions that gather and display statistics about a computation, such as as the elapsed time, and the number of data objects of different types allocated. TIMEALL and TIME gather statistics on the evaluation of a specified form. BREAKDOWN gathers statistics on individual functions called during a computation. These functions can be used to determine which parts of a computation are consuming the most resources (time, storage, etc.), and could most profitably be improved.

(**TIMEALL** *TIMEFORM NUMBEROFTIMES TIMEWHAT INTERPFLG*)   [NLambda Function]

> Evaluates the form *TIMEFORM* and prints statistics on time spent in various categories (elapsed, keyboard wait, swapping time, gc) and data type allocation.

> For more accurate measurement on small computations, *NUMBEROFTIMES* may be specified (its default is 1) to cause *TIMEFORM* to be executed *NUMBEROFTIMES* times. To improve the accuracy of timing open-coded operations in this case, TIMEALL compiles a form to execute *TIMEFORM NUMBEROFTIMES* times (unless *INTERPFLG* is non-NIL), and then times the execution of the compiled form.

>> Note: If TIMEALL is called with *NUMBEROFTIMES* > 1, the dummy form is compiled with compiler optimizations on. This means that it is not meaningful to use TIMEALL with very simple forms that are optimized out by the compiler. For example, (TIMEALL '(IPLUS 2 3) 1000) will time a compiled function which simply returns the number 5, since (IPLUS 2 3) is optimized to the integer 5.

> *TIMEWHAT* restricts the statistics to specific categories. It can be an atom or list of datatypes to monitor, and/or the atom TIME to monitor time spent. Note that ordinarily, TIMEALL monitors all time and datatype usage, so this argument is rarely needed.

> TIMEALL returns the value of the last evaluation of *TIMEFORM*.

(**TIME** *TIMEX TIMEN TIMETYP*)   [NLambda Function]

> TIME evaluates the form *TIMEX*, and prints out the number of CONS cells allocated and computation time. Garbage collection time is subtracted out. This function has been largely replaced by TIMEALL.

> If *TIMEN* is greater than 1, *TIMEX* is executed *TIMEN* times, and TIME prints out (number of conses)/*TIMEN*, and (computation time)/*TIMEN*. If *TIMEN* = NIL, it defaults to 1. This is useful for more accurate measurement on small computations.

> If *TIMETYP* is 0, TIME measures and prints total *real* time as well as computation time. If *TIMETYP* = 3, TIME measures and prints garbage collection time as well as computation time. If *TIMETYP* = T, TIME measures and prints the number of pagefaults.

> TIME returns the value of the last evaluation of *TIMEX*.

(**BOXCOUNT** *TYPE N*)   [Function]

> Returns the number of data objects of type *TYPE* allocated since this Interlisp system was created. *TYPE* can be any data type name (see TYPENAME, Chapter 8). If *TYPE* is NIL, it defaults to FIXP. If *N* is non-NIL, the corresponding counter is reset to *N*.

(**CONSCOUNT** *N*)   [Function]

> Returns the number of CONS cells allocated since this Interlisp system was created. If *N* is non-NIL, resets the counter to *N*. Equivalent to (BOXCOUNT 'LISTP *N*).

(**PAGEFAULTS**) [Function]

Returns the number of page faults since this Interlisp system was created.

BREAKDOWN

TIMEALL collects statistics for whole computations. BREAKDOWN is available to analyze the breakdown of computation time (or any other measureable quantity) function by function.

(**BREAKDOWN** FN_1 ... FN_N) [NLambda NoSpread Function]

You call BREAKDOWN giving it a list of function names (unevaluated). These functions are modified so that they keep track of various statistics.

To remove functions from those being monitored, simply UNBREAK (Chapter 15) the functions, thereby restoring them to their original state. To add functions, call BREAKDOWN on the new functions. This will not reset the counters for any functions not on the new list. However (BREAKDOWN) will zero the counters of all functions being monitored.

The procedure used for measuring is such that if one function calls other and both are "broken down", then the time (or whatever quantity is being measured) spent in the inner function is *not* charged to the outer function as well.

BREAKDOWN will *not* give accurate results if a function being measured is not returned from normally, e.g., a lower RETFROM (or ERROR) bypasses it. In this case, all of the time (or whatever quantity is being measured) between the time that function is entered and the time the next function being measured is entered will be charged to the first function.

(**BRKDWNRESULTS** *RETURNVALUESFLG*) [Function]

BRKDWNRESULTS prints the analysis of the statistics requested as well as the number of calls to each function. If *RETURNVALUESFLG* is non-NIL, BRKDWNRESULTS will not to print the results, but instead return them in the form of a list of elements of the form (*FNNAME #CALLS VALUE*).

Example:

```
← (BREAKDOWN SUPERPRINT SUBPRINT COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
←(PRETTYDEF '(SUPERPRINT) 'FOO)
FOO.;3
←(BRKDWNRESULTS)
FUNCTIONS    TIME    #CALLS   PER CALL    %
SUPERPRINT  8.261    365     0.023      20
SUBPRINT   31.910    141     0.226      76
COMMENT1    1.612      8     0.201       4
TOTAL      41.783    514     0.081
NIL
←(BRKDWNRESULTS T)
```

```
((SUPERPRINT 365 8261) (SUBPRINT 141 31910)
(COMMENT1 8 1612))
```

BREAKDOWN can be used to measure other statistics, by setting the following variables:

BRKDWNTYPE   [Variable]

To use BREAKDOWN to measure other statistics, before calling BREAKDOWN, set the variable BRKDWNTYPE to the quantity of interest, e.g., TIME, CONSES, etc, or a list of such quantities. Whenever BREAKDOWN is called with BRKDWNTYPE not NIL, BREAKDOWN performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with a particular statistic, a measuring function will be defined, and the compiler will be called to compile it. The functions being broken down will be redefined to call this measuring function. When BREAKDOWN is through initializing, it sets BRKDWNTYPE back to NIL. Subsequent calls to BREAKDOWN will measure the new statistic until BRKDWNTYPE is again set and a new BREAKDOWN performed.

BRKDWNTYPES   [Variable]

The list BRKDWNTYPES contains the information used to analyze new statistics. Each entry on BRKDWNTYPES should be of the form (*TYPE FORM FUNCTION*), where *TYPE* is a statistic name (as would appear in BRKDWNTYPE), *FORM* computes the statistic, and *FUNCTION* (optional) converts the value of form to some more interesting quantity. For example, (TIME (CLOCK 2) (LAMBDA (X) (FQUOTIENT X 1000))) measures computation time and reports the result in seconds instead of milliseconds. BRKDWNTYPES currently contains entries for TIME, CONSES, PAGEFAULTS, BOXES, and FBOXES.

Example:

```
←(SETQ BRKDWNTYPE '(TIME CONSES))
(TIME CONSES)
←(BREAKDOWN MATCH CONSTRUCT)
(MATCH CONSTRUCT)
←(FLIP '(A B C D E F G H C Z) '(.. $1 .. #2 ..)
'(.. #3 ..))
(A B D E F G H Z)
←(BRKDWNRESULTS)
FUNCTIONS  TIME    #CALLS  PER CALL   %
MATCH      0.036   1         0.036    54
CONSTRUCT  0.031   1         0.031    46
TOTAL      0.067   2         0.033
FUNCTIONS  CONSES  #CALLS  PER CALL   %
MATCH      32      1        32.000    40
CONSTRUCT  49      1        49.000    60
TOTAL      81      2        40.500
NIL
```

Occasionally, a function being analyzed is sufficiently fast that the overhead involved in measuring it obscures the actual time spent in the function. If you were using TIME, you would specify a value for *TIMEN* greater than 1 to give greater accuracy. A similar option is available for BREAKDOWN. You can specify that a function(s) be executed a multiple

number of times for each measurement, and the average value reported, by including a number in the list of functions given to BREAKDOWN. For example, BREAKDOWN(EDITCOM EDIT4F 10 EDIT4E EQP) means normal breakdown for EDITCOM and EDIT4F but executes (the body of) EDIT4E and EQP 10 times each time they are called. Of course, the functions so measured must not cause any harmful side effects, since they are executed more than once for each call. The printout from BRKDWNRESULTS will look the same as though each function were run only once, except that the measurement will be more accurate.

Another way of obtaining more accurate measurement is to expand the call to the measuring function in-line. If the value of BRKDWNCOMPFLG is non-NIL (initially NIL), then whenever a function is broken-down, it will be redefined to call the measuring function, and then recompiled. The measuring function is expanded in-line via an appropriate macro. In addition, whenever BRKDWNTYPE is reset, the compiler is called for *all* functions for which BRKDWNCOMPFLG was set at the time they were originally broken-down, i.e. the setting of the flag at the time a function is broken-down determines whether the call to the measuring code is compiled in-line.

GAINSPACE

If you have large programs and databases, you may sometimes find yourself in a situation where you need to obtain more space, and are willing to pay the price of eliminating some or all of the context information that the various user-assistance facilities such as the programmer's assistant, file package, CLISP, etc., have accumulated during the course of his session. The function GAINSPACE provides an easy way to selectively throw away accumulated data:

(**GAINSPACE**)   [Function]

> Prints a list of deletable objects, allowing you to specify at each point what should be discarded and what should be retained. For example:

```
←(GAINSPACE)
purge history lists ? Yes
purge everything, or just the properties, e.g.,
SIDE, LISPXPRINT, etc. ?
just the properties
discard definitions on property lists ? Yes
discard old values of variables ? Yes
erase properties ? No
erase CLISP translations? Yes
```

GAINSPACE is driven by the list GAINSPACEFORMS. Each element on GAINSPACEFORMS is of the form (*PRECHECK MESSAGE FORM KEYLST*). If *PRECHECK*, when evaluated, returns NIL, GAINSPACE skips to the next entry. For example, you will not be asked whether or not to purge the history list if it is not enabled. Otherwise, ASKUSER (Chapter 26) is called with the indicated *MESSAGE* and the (optional) *KEYLST*. If you respond No, i.e., ASKUSER returns N, GAINSPACE skips to the next entry. Otherwise, *FORM* is evaluated with the variable RESPONSE bound to the value of ASKUSER. In the above example, the *FORM* for the "purge history lists" question calls ASKUSER to ask

PERFORMANCE ISSUES

"`purge everything, ...`" only if you had responded `Yes`. If you had responded with `Everything`, the second question would not have been asked.

The "`erase properties`" question is driven by a list SMASHPROPSMENU. Each element on this list is of the form (*MESSAGE . PROPS*). You are prompted with *MESSAGE* (by ASKUSER), and if your response is `Yes`, *PROPS* is added to the list SMASHPROPS. The "`discard definitions on property lists`" and "`discard old values of variables`" questions also add to SMASHPROPS. You will not be prompted for any entry on SMASHPROPSMENU for which all of the corresponding properties are already on SMASHPROPS. SMASHPROPS is initially set to the value of SMASHPROPSLST. This permits you to specify in advance those properties which you always want discarded, and not be asked about them subsequently. After finishing all the entries on GAINSPACEFORMS, GAINSPACE checks to see if the value of SMASHPROPS is non-NIL, and if so, does a MAPATOMS, i.e., looks at every atom in the system, and erases the indicated properties.

You can change or add new entries to GAINSPACEFORMS or SMASHPROPSMENU, so that GAINSPACE can also be used to purge structures that your programs have accumulated.

Using Data Types Instead of Records

If a program uses large numbers of large data structures, there are several advantages to representing them as user data types rather than as list structures. The primary advantage is increased speed: accessing and setting the fields of a data type can be significantly faster than walking through a list with repeated CARs and CDRs. Also,

Compiled code for referencing data types is usually smaller. Finally, by reducing the number of objects created (one object against many list cells), this can reduce the expense of garbage collection.

User data types are declared by using the DATATYPE record type (Chapter 8). If a list structure has been defined using the RECORD record type (Chapter 8), and all accessing operations are written using the record package's `fetch`, `replace`, and `create` operations, changing from RECORDs to DATATYPEs only requires editing the record declaration (using EDITREC, Chapter 8) to replace declaration type RECORD by DATATYPE, and recompiling.

> Note: There are some minor disadvantages: First, there is an upper limit on the number of data types that can exist. Also, space for data types is allocated two pages at a time. Each data type which has any instances allocated has at least two pages assigned to it, which may be wasteful of space if there are only a few examples of a given data type. These problems should not effect most applications programs.

21-11

Using "Fast" and "Destructive" Functions

Among the functions used for manipulating objects of various data types, there are a number of functions which have "fast" and "destructive" versions. You should be aware of what these functions do, and when they should be used.

"Fast" functions: By convention, a function named by prefixing an existing function name with F indicates that the new function is a "fast" version of the old. These usually have the same definitions as the slower versions, but they compile open and run without any "safety" error checks. For example, FNTH runs faster than NTH, however, it does not make as many checks (for lists ending with anything but NIL, etc). If these functions are given arguments that are not in the form that they expect, their behavior is unpredictable; they may run forever, or cause a system error. In general, you should only use "fast" functions in code that has already been completely debugged, to speed it up.

"Destructive" functions: By convention, a function named by prefixing an existing function with D indicates the new function is a "destructive" version of the old one, which does not make any new structure but cannibalizes its argument(s). For example, REMOVE returns a copy of a list with a particular element removed, but DREMOVE actually changes the list structure of the list. (Unfortunately, not all destructive functions follow this naming convention: the destructive version of APPEND is NCONC.) You should be careful when using destructive functions that they do not inadvertantly change data structures.

22. PERFORMANCE ISSUES

This chapter describes a number of areas that often contribute to performance problems in Interlisp-D programs. Many performance problems can be improved by optimizing the use of storage, since allocating and reclaiming large amounts of storage is expensive. Another tactic that can sometimes yield performance improvements is to change the use of variable bindings on the stack to reduce variable lookup time. There are a number of tools that can be used to determine which parts of a computation cause performance bottlenecks.

Storage Allocation and Garbage Collection

As an Interlisp-D applications program runs, it creates data structures (allocated out of free storage space), manipulates them, and then discards them. If there were no way of reclaiming this space, over time the Interlisp-D memory (both the physical memory in the machine and the virtual memory stored on the disk) would fill up, and the computation would come to a halt. Actually, long before this could happen the system would probably become intolerably slow, due to "data fragmentation," which occurs when the data currently in use are spread over many virtual memory pages, so that most of the computer time must be spent swapping disk pages into physical memory. The problem of fragmentation will occur in any situation where the virtual memory is significantly larger than the real physical memory. To reduce swapping, it is desirable to keep the "working set" (the set of pages containing actively referenced data) as small as possible.

It is possible to write programs that don't generate much "garbage" data, or which recycle data, but such programs tend to be overly complicated and difficult to debug. Spending effort writing such programs defeats the whole point of using a system with automatic storage allocation. An important part of any Lisp implementation is the "garbage collector" which identifies discarded data and reclaims its space. There are several well-known approaches to garbage collection. One method is the traditional mark-and-sweep garbage collection algorithm, which identifies "garbage" data by marking all accessible data structures, and then sweeping through the data spaces to find all unmarked objects (i.e., not referenced by any other object).   Although this method is guaranteed to reclaim all garbage, it takes time proportional to the number of allocated objects, which may be very large. (Some allocated objects will have been marked during the "mark" phase, and the remainder will be collected during the "sweep" phase; so all will have to be touched in some way.) Also, the time that a mark-and-sweep garbage collection takes is independent of the amount of garbage collected; it

is possible to sweep through the whole virtual memory, and only recover a small amount of garbage.

For interactive applications, it is not acceptable to have long interruptions in a computation for the purpose of garbage collection. Interlisp-D solves this problem by using a reference-counting garbage collector. With this scheme, there is a table containing counts of how many times each object is referenced. This table is incrementally updated as pointers are created and discarded, incurring a small overhead distributed over the computation as a whole. (Note: References from the stack are not counted, but are handled separately at "sweep" time; thus the vast majority of data manipulations do not cause updates to this table.) At opportune moments, the garbage collector scans this table, and reclaims all objects that are no longer accessible (have a reference count of zero). The pause while objects are reclaimed is only the time for scanning the reference count tables (small) plus time proportional to the amount of garbage that has to be collected (typically less than a second). "Opportune" times occur when a certain number of cells have been allocated or when the system has been waiting for the user to type something for long enough. The frequency of garbage collection is controlled by the functions and variables described below. For the best system performance, it is desirable to adjust these parameters for frequent, short garbage collections, which will not interrupt interactive applications for very long, and which will have the added benefit of reducing data fragmentation, keeping the working set small.

One problem with the Interlisp-D garbage collector is that not all garbage is guaranteed to be collected. Circular data structures, which point to themselves directly or indirectly, are never reclaimed, since their reference counts are always at least one. With time, this unreclaimable garbage may increase the working set to unacceptable levels. Some users have worked with the same Interlisp-D virtual memory for a very long time, but it is a good idea to occasionally save all of your functions in files, reinitialize Interlisp-D, and rebuild your system. Many users end their working day by issuing a command to rebuild their system and then leaving the machine to perform this task in their absence. If the system seems to be spending too much time swapping (an indication of fragmented working set), this procedure is definitely recommended.

Garbage collection in Interlisp-D is controlled by the following functions and variables:

(RECLAIM)   [Function]

      Initiates a garbage collection. Returns 0.

(RECLAIMMIN N)   [Function]

      Sets the frequency of garbage collection. Interlisp keeps track of the number of cells of any type that have been allocated; when it reaches a given number, a garbage collection occurs. If N is non-NIL, this number is set to N. Returns the current setting of the number.

RECLAIMWAIT   [Variable]

     Interlisp-D will invoke a `RECLAIM` if the system is idle and waiting for your
     input for `RECLAIMWAIT` seconds (currently set for 4 seconds).

(GCGAG *MESSAGE*)   [Function]

     Sets the behavior that occurs while a garbage collection is taking place. If
     MESSAGE is non-`NIL`, the cursor is complemented during a `RECLAIM`; if
     MESSAGE=`NIL`, nothing happens. The value of `GCGAG` is its previous setting.

(GCTRP)   [Function]

     Returns the number of cells until the next garbage collection, according to the
     `RECLAIMMIN` number.

The amount of storage allocated to different data types, how much of that storage is in
use, and the amount of data fragmentation can be determined using the following
function:

(STORAGE *TYPES PAGETHRESHOLD*)   [Function]

     `STORAGE` prints out a summary, for each data type, of the amount of space
     allocated to the data type, and how much of that space is currently in use. If
     TYPES is non-`NIL`, `STORAGE` only lists statistics for the specified types. *TYPES*
     can be a litatom or a list of types. If *PAGETHRESHOLD* is non-`NIL`, then
     `STORAGE` only lists statistics for types that have at least *PAGETHRESHOLD*
     pages allocated to them.

STORAGE prints out a table with the column headings **Type**, **Assigned**, **Free Items**, **In use**, and **Total alloc**. **Type** is the name of the data type. **Assigned** is how much of your virtual memory is set aside for items of this type. Currently, memory is allocated in quanta of two pages (1024 bytes). The numbers under **Assigned** show the number of pages and the total number of items that fit on those pages. **Free Items** shows how many items are available to be allocated (using the **create** construct, Chapter 8); these constitute the "free list" for that data type. **In use** shows how many items of this type are currently in use, i.e., have pointers to them and hence have not been garbage collected. If this number is higher than your program seems to warrant, you may want to look for storage leaks. The sum of **Free Items** and **In use** is always the same as the total **Assigned** items. **Total alloc** is the total number of items of this type that have ever been allocated (see BOXCOUNT, in the Performance Measuring section below).

Note: The information about the number of items of type LISTP is only approximate, because list cells are allocated in a special way that precludes easy computation of the number of items per page.

Note: When a data type is redeclared, the data type name is reassigned. Pages which were assigned to instances of the old data type are labeled ****DEALLOC****.

At the end of the table printout, STORAGE prints a "Data Spaces Summary" listing the number of pages allocated to the major data areas in the virtual address space: the space for fixed-length items (including datatypes), the space for variable-length items, and the space for litatoms. Variable-length data types such as arrays have fixed-length "headers," which is why they also appear in the printout of fixed-length data types. Thus, the line printed for the BITMAP data type says how many bitmaps have been allocated, but the "assigned pages" column counts only the headers, not the space used by the variable-length part of the bitmap. This summary also lists "Remaining Pages" in relation to the largest possible virtual memory, not the size of the virtual memory backing file in use. This file may fill up, causing a **STORAGE FULL** error, long before the "Remaining Pages" numbers reach zero.

STORAGE also prints out information about the sizes of the entries on the variable-length data free list. The block sizes are broken down by the value of the variable STORAGE.ARRAYSIZES, initially (4 16 64 256 1024 4096 16384 NIL), which yields a printout of the form:

```
variable-datum free list:
le 4        26 items;     104 cells.
le 16       72 items;     783 cells.
le 64       36 items;     964 cells.
le 256      28 items;    3155 cells.
le 1024      3 items;    1175 cells.
```

```
le 4096      5 items;   8303 cells.
le 16384     3 items;  17067 cells.
others       1 items;  17559 cells.
```

This information can be useful in determining if the variable-length data space is fragmented. If most of the free space is composed of small items, then the allocator may not be able to find room for large items, and will extend the variable datum space. If this is extended too much, this could cause an ARRAYS FULL error, even if there is a lot of space left in little chunks.

`(STORAGE.LEFT)` [Function]

Provides a programmatic way of determining how much storage is left in the major data areas in the virtual address space. Returns a list of the form (*MDSFREE MDSFRAC 8MBFRAC ATOMFREE ATOMFRAC*), where the elements are interpreted as follows:

MDSFREE	The number of free pages left in the main data space (which includes both fixed-length and variable-length data types).
MDSFRAC	The fraction of the total possible main data space that is free.
8MBFRAC	The fraction of the total main data space that is free, relative to eight megabytes.
	This number is useful when using Interlisp-D on some early computers where the hardware limits the address space to eight megabytes. The function `32MBADDRESSABLE` returns non-`NIL` if the currently running Interlisp-D system can use the full 32 megabyte address space.
ATOMFREE	The number of free pages left in the litatom space.
ATOMFRAC	The fraction of the total litatom space that is free.

Note: Another important space resource is the amount of the virtual memory backing file in use (see `VMEMSIZE`, Chapter 12). The system will crash if the virtual memory file is full, even if the address space is not exhausted.

Variable Bindings

Different implementations of lisp use different methods of accessing free variables. The binding of variables occurs when a function or a `PROG` is entered. For example, if the function `FOO` has the definition (`LAMBDA (A B)` *BODY*), the variables `A` and `B` are bound so that any reference to `A` or `B` from *BODY* or any function called from *BODY* will refer to the arguments to the function `FOO` and not to the value of `A` or `B` from a higher level function. All variable names (litatoms) have a top level value cell which is used if the variable has not been bound in any function. In discussions of variable access, it is useful to distinquish between three types of variable access: local, special and global. Local variable access is the use of a variable that is bound within the function from which it is used. Special variable access is the use of a variable that is bound by another function. Global variable access is the use of a variable that has not been bound in any function. We will often refer to a variable all of whose accesses are local as a "local variable." Similarly, a variable all of whose accesses are global we call a "global variable."

In a "deep" bound system, a variable is bound by saving on the stack the variable's name together with a value cell which contains that variable's new value. When a variable is accessed, its value is found by searching the stack for the most recent binding (occurrence) and retrieving the value stored there. If the variable is not found on the stack, the variable's top level value cell is used.

In a "shallow" bound system, a variable is bound by saving on the stack the variable name and the variable's old value and putting the new value in the variable's top level value cell. When a variable is accessed, its value is always found in its top level value cell.

The deep binding scheme has one disadvantage: the amount of cpu time required to fetch the value of a variable depends on the stack distance between its use and its binding. The compiler can determine local variable accesses and compiles them as fetches directly from the stack. Thus this computation cost only arises in the use of variable not bound in the local frame ("free" variables). The process of finding the value of a free variable is called free variable lookup.

In a shallow bound system, the amount of cpu time required to fetch the value of a variable is constant regardless of whether the variable is local, special or global. The disadvantages of this scheme are that the actual binding of a variable takes longer (thus slowing down function call), the cells that contain the current in use values are spread throughout the space of all litatom value cells (thus increasing the working set size of functions) and context switching between processes requires unwinding and rewinding the stack (thus effectively prohibiting the use of context switching for many applications).

Interlisp-D uses deep binding, because of the working set considerations and the speed of context switching. The free variable lookup routine is microcoded, thus greatly reducing the search time. In benchmarks, the largest percentage of free variable lookup time was 20 percent of the total ellapsed time; the normal time was between 5 and 10 percent.

One consequence of Interlisp-D's deep binding scheme is that users may significantly improve performance by declaring global variables in certain situations. If a variable is declared global, the compiler will compile an access to that variable as a retrieval of its top level value, completely bypassing a stack search. This should be done only for variables that are never bound in functions, such as global databases and flags.

Global variable declarations should be done using the GLOBALVARS file package command (Chapter 17). Its form is (GLOBALVARS $VAR_1 \ldots VAR_N$).

Another way of improving performance is to declare variables as local within a function. Normally, all variables bound within a function have their names put on the stack, and these names are scanned during free variable lookup. If a variable is declared to be local within a function, its name is not put on the stack, so it is not scanned during free variable lookup, which may increase the speed of lookups. The compiler can also make some other optimizations if a variable is known to be local to a function.

A variable may be declared as local within a function by including the form (DECLARE (LOCALVARS $VAR_1 \ldots VAR_N$)) following the argument list in the definition of the function. Local variable declarations only effect the compilation of a function. Interpreted functions put all of their variable names on the stack, regardless of any declarations.

Performance Measuring

This section describes functions that gather and display statistics about a computation, such as as the elapsed time, and the number of data objects of different types allocated. `TIMEALL` and `TIME` gather statistics on the evaluation of a specified form. `BREAKDOWN` gathers statistics on individual functions called during a computation. These functions can be used to determine which parts of a computation are consuming the most resources (time, storage, etc.), and could most profitably be improved.

(`TIMEALL` *TIMEFORM NUMBEROFTIMES TIMEWHAT INTERPFLG* —)
　　[NLambda Function]

> Evaluates the form *TIMEFORM* and prints statistics on time spent in various categories (elapsed, keyboard wait, swapping time, gc) and data type allocation.

> For more accurate measurement on small computations, *NUMBEROFTIMES* may be specified (its default is 1) to cause *TIMEFORM* to be executed *NUMBEROFTIMES* times. To improve the accuracy of timing open-coded operations in this case, `TIMEALL` compiles a form to execute *TIMEFORM* *NUMBEROFTIMES* times (unless *INTERPFLG* is non-`NIL`), and then times the execution of the compiled form.

> Note:　If `TIMEALL` is called with *NUMBEROFTIMES*>1, the dummy form is compiled with compiler optimizations on. This means that it is not meaningful to use `TIMEALL` with very simple forms that are optimized out by the compiler. For example, (`TIMEALL '(IPLUS 2 3) 1000`) will time a compiled function which simply returns the number 5, since (`IPLUS 2 3`) is optimized to the integer 5.

> *TIMEWHAT* restricts the statistics to specific categories. It can be an atom or list of datatypes to monitor, and/or the atom `TIME` to monitor time spent. Note that ordinarily, `TIMEALL` monitors all time and datatype usage, so this argument is rarely needed.

> `TIMEALL` returns the value of the last evaluation of *TIMEFORM*.

(`TIME` *TIMEX TIMEN TIMETYP*)　　　　　　　　　　　　　[NLambda Function]

> `TIME` evaluates the form *TIMEX*, and prints out the number of `CONS` cells allocated and computation time. Garbage collection time is subtracted out. This function has been largely replaced by `TIMEALL`.

> If *TIMEN* is greater than 1, *TIMEX* is executed *TIMEN* times, and `TIME` prints out (number of conses)/*TIMEN*, and (computation time)/*TIMEN*. If *TIMEN*=`NIL`, it defaults to 1. This is useful for more accurate measurement on small computations.

If *TIMETYP* is 0, `TIME` measures and prints total *real* time as well as computation time. If *TIMETYP* = 3, `TIME` measures and prints garbage collection time as well as computation time. If *TIMETYP*=T, `TIME` measures and prints the number of pagefaults.

`TIME` returns the value of the last evaluation of *TIMEX*.

(`BOXCOUNT` *TYPE N*)   [Function]

Returns the number of data objects of type *TYPE* allocated since this Interlisp system was created. *TYPE* can be any data type name (see `TYPENAME`, Chapter 8). If *TYPE* is `NIL`, it defaults to `FIXP`. If *N* is non-`NIL`, the corresponding counter is reset to *N*.

(`CONSCOUNT` *N*)   [Function]

Returns the number of `CONS` cells allocated since this Interlisp system was created. If *N* is non-`NIL`, resets the counter to *N*. Equivalent to (`BOXCOUNT` 'LISTP *N*).

(`PAGEFAULTS`)   [Function]

Returns the number of page faults since this Interlisp system was created.

BREAKDOWN

`TIMEALL` collects statistics for whole computations. `BREAKDOWN` is available to analyze the breakdown of computation time (or any other measureable quantity) function by function.

(`BREAKDOWN` *FN$_1$* ... *FN$_N$*)   [NLambda NoSpread Function]

The user calls `BREAKDOWN` giving it a list of function names (unevaluated). These functions are modified so that they keep track of various statistics.

To remove functions from those being monitored, simply `UNBREAK` (Chapter 15) the functions, thereby restoring them to their original state. To add functions, call `BREAKDOWN` on the new functions. This will not reset the counters for any functions not on the new list. However (`BREAKDOWN`) will zero the counters of all functions being monitored.

The procedure used for measuring is such that if one function calls other and both are "broken down", then the time (or whatever quantity is being measured) spent in the inner function is *not* charged to the outer function as well.

BREAKDOWN will *not* give accurate results if a function being measured is not returned from normally, e.g., a lower RETFROM (or ERROR) bypasses it. In this case, all of the time (or whatever quantity is being measured) between the time that function is entered and the time the next function being measured is entered will be charged to the first function.

(BRKDWNRESULTS *RETURNVALUESFLG*)   [Function]

> BRKDWNRESULTS prints the analysis of the statistics requested as well as the number of calls to each function. If *RETURNVALUESFLG* is non-NIL, BRKDWNRESULTS will not to print the results, but instead return them in the form of a list of elements of the form (*FNNAME #CALLS VALUE*).

> Example:

```
← (BREAKDOWN SUPERPRINT SUBPRINT COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
←(PRETTYDEF '(SUPERPRINT) 'FOO)
FOO.;3
←(BRKDWNRESULTS)
FUNCTIONS    TIME      #CALLS   PER CALL    %
SUPERPRINT  8.261      365      0.023      20
SUBPRINT    31.910     141      0.226      76
COMMENT1    1.612       8       0.201       4
TOTAL       41.783     514      0.081
NIL
←(BRKDWNRESULTS T)
((SUPERPRINT 365 8261) (SUBPRINT 141 31910) (COMMENT1 8
1612))
```

BREAKDOWN can be used to measure other statistics, by setting the following variables:

BRKDWNTYPE   [Variable]

> To use BREAKDOWN to measure other statistics, before calling BREAKDOWN, set the variable BRKDWNTYPE to the quantity of interest, e.g., TIME, CONSES, etc, or a list of such quantities. Whenever BREAKDOWN is called with BRKDWNTYPE not NIL, BREAKDOWN performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with a particular statistic, a measuring function will be defined, and the compiler will be called to compile it. The functions being broken down will be redefined to call this measuring function. When BREAKDOWN is through initializing, it sets BRKDWNTYPE back to NIL. Subsequent calls to BREAKDOWN will measure the new statistic until BRKDWNTYPE is again set and a new BREAKDOWN performed.

BRKDWNTYPES   [Variable]

The list BRKDWNTYPES contains the information used to analyze new statistics.
Each entry on BRKDWNTYPES should be of the form (*TYPE FORM FUNCTION*),
where *TYPE* is a statistic name (as would appear in BRKDWNTYPE), *FORM*
computes the statistic, and *FUNCTION* (optional) converts the value of form to
some more interesting quantity. For example, (TIME (CLOCK 2) (LAMBDA
(X) (FQUOTIENT X 1000))) measures computation time and reports the
result in seconds instead of milliseconds. BRKDWNTYPES currently contains
entries for TIME, CONSES, PAGEFAULTS, BOXES, and FBOXES.

Example:

```
←(SETQ BRKDWNTYPE '(TIME CONSES))
(TIME CONSES)
←(BREAKDOWN MATCH CONSTRUCT)
(MATCH CONSTRUCT)
←(FLIP '(A B C D E F G H C Z) '(.. $1 .. #2 ..) '(.. #3
..))
(A B D E F G H Z)
←(BRKDWNRESULTS)
FUNCTIONS   TIME      #CALLS   PER CALL    %
MATCH       0.036     1           0.036    54
CONSTRUCT   0.031     1           0.031    46
TOTAL       0.067     2           0.033
FUNCTIONS   CONSES    #CALLS   PER CALL    %
MATCH       32        1          32.000    40
CONSTRUCT   49        1          49.000    60
TOTAL       81        2          40.500
NIL
```

Occasionally, a function being analyzed is sufficiently fast that the overhead
involved in measuring it obscures the actual time spent in the function. If you
were using TIME, you would specify a value for *TIMEN* greater than 1 to give
greater accuracy. A similar option is available for BREAKDOWN. You can specify
that a function(s) be executed a multiple number of times for each
measurement, and the average value reported, by including a number in the list
of functions given to BREAKDOWN. For example, BREAKDOWN(EDITCOM EDIT4F
10 EDIT4E EQP) means normal breakdown for EDITCOM and EDIT4F but
executes (the body of) EDIT4E and EQP 10 times each time they are called. Of
course, the functions so measured must not cause any harmful side effects, since
they are executed more than once for each call. The printout from
BRKDWNRESULTS will look the same as though each function were run only once,
except that the measurement will be more accurate.

Another way of obtaining more accurate measurement is to expand the call to
the measuring function in-line. If the value of BRKDWNCOMPFLG is non-NIL
(initially NIL), then whenever a function is broken-down, it will be redefined to

call the measuring function, and then recompiled. The measuring function is expanded in-line via an appropriate macro. In addition, whenever BRKDWNTYPE is reset, the compiler is called for *all* functions for which BRKDWNCOMPFLG was set at the time they were originally broken-down, i.e. the setting of the flag at the time a function is broken-down determines whether the call to the measuring code is compiled in-line.

GAINSPACE

If you have large programs and databases, you may sometimes find yourself in a situation where you need to obtain more space, and are willing to pay the price of eliminating some or all of the context information that the various user-assistance facilities such as the programmer's assistant, file package, CLISP, etc., have accumulated during the course of his session. The function GAINSPACE provides an easy way to selectively throw away accumulated data:

(GAINSPACE) [Function]

Prints a list of deletable objects, allowing you to specify at each point what should be discarded and what should be retained. For example:

```
←(GAINSPACE)
purge history lists ? Yes
purge everything, or just the properties, e.g., SIDE,
LISPXPRINT, etc. ?
just the properties
discard definitions on property lists ? Yes
discard old values of variables ? Yes
erase properties ? No
erase CLISP translations? Yes
```

GAINSPACE is driven by the list GAINSPACEFORMS. Each element on GAINSPACEFORMS is of the form (*PRECHECK MESSAGE FORM KEYLST*). If *PRECHECK*, when evaluated, returns NIL, GAINSPACE skips to the next entry. For example, you will not be asked whether or not to purge the history list if it is not enabled. Otherwise, ASKUSER (Chapter 26) is called with the indicated *MESSAGE* and the (optional) *KEYLST*. If you respond **No**, i.e., ASKUSER returns N, GAINSPACE skips to the next entry. Otherwise, *FORM* is evaluated with the variable RESPONSE bound to the value of ASKUSER. In the above example, the *FORM* for the "**purge history lists**" question calls ASKUSER to ask "**purge everything, ...**" only if you had responded **Yes**. If you had responded with **Everything**, the second question would not have been asked.

The "**erase properties**" question is driven by a list SMASHPROPSMENU. Each element on this list is of the form (*MESSAGE . PROPS*). You are prompted with *MESSAGE*

(by ASKUSER), and if your response is **Yes**, *PROPS* is added to the list SMASHPROPS. The "**discard definitions on property lists**" and "**discard old values of variables**" questions also add to SMASHPROPS. You will not be prompted for any entry on SMASHPROPSMENU for which all of the corresponding properties are already on SMASHPROPS. SMASHPROPS is initially set to the value of SMASHPROPSLST. This permits you to specify in advance those properties which you always want discarded, and not be asked about them subsequently. After finishing all the entries on GAINSPACEFORMS, GAINSPACE checks to see if the value of SMASHPROPS is non-NIL, and if so, does a MAPATOMS, i.e., looks at every atom in the system, and erases the indicated properties.

You can change or add new entries to GAINSPACEFORMS or SMASHPROPSMENU, so that GAINSPACE can also be used to purge structures that your programs have accumulated.

Using Data Types Instead of Records

If a program uses large numbers of large data structures, there are several advantages to representing them as user data types rather than as list structures. The primary advantage is increased speed: accessing and setting the fields of a data type can be significantly faster than walking through a list with repeated CARs and CDRs. Also,

compiled code for referencing data types is usually smaller. Finally, by reducing the number of objects created (one object against many list cells), this can reduce the expense of garbage collection.

User data types are declared by using the `DATATYPE` record type (Chapter 8). If a list structure has been defined using the `RECORD` record type (Chapter 8), and all accessing operations are written using the record package's **fetch**, **replace**, and **create** operations, changing from `RECORD`s to `DATATYPE`s only requires editing the record declaration (using `EDITREC`, Chapter 8) to replace declaration type `RECORD` by `DATATYPE`, and recompiling.

Note: There are some minor disadvantages with allocating new data types: First, there is an upper limit on the number of data types which can exist. Also, space for data types is allocated a page at a time, so each data type has at least one page assigned to it, which may be wasteful of space if there are only a few examples of a given data type. These problems should not effect most applications programs.

Using Incomplete File Names

Currently, Interlisp allows you to specify an open file by giving the file name. If the file name is incomplete (it doesn't have the device/host, directory, name, extension, and version number all supplied), the system converts it to a complete file name, by supplying defaults and searching through directories (which may be on remote file servers), and then searches the open streams for one corresponding to that file name. This file name-completion process happens whenever any I/O function is given an incomplete file name, which can cause a serious performance problem if I/O operations are done repeatedly. In general, it is much faster to convert an incomplete file name to a stream once, and use the stream from then on. For example, suppose a file is opened with `(SETQ STRM (OPENSTREAM 'MYNAME 'INPUT))`. After doing this, `(READC 'MYNAME)` and `(READC STRM)` would both work, but `(READC 'MYNAME)` would take longer (sometimes orders of magnitude longer). This could seriously effect the performance if a program which is doing many I/O operations.

At some point in the future, when multiple streams are supported to a single file, the feature of mapping file names to streams will be removed. This is yet another reason why programs should use streams as handles to open files, instead of file names.

For more information on efficiency considerations when using files, see Chapter 24.

Using "Fast" and "Destructive" Functions

Among the functions used for manipulating objects of various data types, there are a number of functions which have "fast" and "destructive" versions. You should be aware of what these functions do, and when they should be used.

"Fast" functions: By convention, a function named by prefixing an existing function name with F indicates that the new function is a "fast" version of the old. These usually have the same definitions as the slower versions, but they compile open and run without any "safety" error checks. For example, FNTH runs faster than NTH, however, it does not make as many checks (for lists ending with anything but NIL, etc). If these functions are given arguments that are not in the form that they expect, their behavior is unpredictable; they may run forever, or cause a system error. In general, you should only use "fast" functions in code that has already been completely debugged, to speed it up.

"Destructive" functions: By convention, a function named by prefixing an existing function with `D` indicates the new function is a "destructive" version of the old one, which does not make any new structure but cannibalizes its argument(s). For example, `REMOVE` returns a copy of a list with a particular element removed, but `DREMOVE` actually changes the list structure of the list. (Unfortunately, not all destructive functions follow this naming convention: the destructive version of `APPEND` is `NCONC`.) You should be careful when using destructive functions that they do not inadvertantly change data structures.

22. PROCESSES

The Medley Process mechanism provides an environment in which multiple Lisp processes can run in parallel. Each executes in its own stack space, but all share a global address space. The current process implementation is cooperative; i.e., process switches happen voluntarily, either when the process in control has nothing to do or when it is in a convenient place to pause. There is no preemption or guaranteed service, so you cannot run something demanding (e.g., Chat) at the same time as something that runs for long periods without yielding control. Keyboard input and network operations block with great frequency, so processes currently work best for highly interactive tasks (editing, making remote files).

In Medley, the process mechanism is already turned on, and is expected to stay on during normal operations, as some system facilities (in particular, most network operations) require it. However, under exceptional conditions, the following function can be used to turn the world off and on:

> (**PROCESSWORLD** *FLG*)   [Function]
>
> Starts up the process world, or if *FLG* = OFF, kills all processes and turns it off. Normally does not return. The environment starts out with two processes: a top-level EVALQT (the initial "tty" process) and the "background" process, which runs the window mouse handler and other system background tasks.
>
> PROCESSWORLD is intended to be called at the top level of Interlisp, not from within a program. It does not toggle some sort of switch; rather, it constructs some new processes in a new part of the stack, leaving any callers of PROCESSWORLD in a now inaccessible part of the stack. Calling (PROCESSWORLD 'OFF) is the only way the call to PROCESSWORLD ever returns.

> (**HARDRESET**)   [Function]
>
> Resets the whole world, and rebuilds the stack from scratch. This is "harder" than doing RESET to every process, because it also resets system internal processes (such as the keyboard handler).
>
> HARDRESET automatically turns the process world on (or resets it if it was on), unless the variable AUTOPROCESSFLG is NIL.

Creating and Destroying Processes

> (**ADD.PROCESS** *FORM PROP$_1$ VALUE$_1$ ... PROP$_N$ VALUE$_N$*)   [NoSpread Function]
>
> Creates a new process evaluating *FORM*, and returns its process handle. The process's stack environment is the top level, i.e., the new process does not have access to the environment in which ADD.PROCESS was called; all such information must be passed as arguments in *FORM*. The process runs until *FORM* returns or the process is explicitly deleted. An untrapped error within the process also deletes the process (unless its RESTARTABLE property is T), in which case a message is printed to that effect.

The remaining arguments are alternately property names and values. Any property/value pairs acceptable to PROCESSPROP may be given, but the following two are directly relevant to ADD.PROCESS:

NAME	Value can be a symbol or a string; if not given, the process name is taken from (CAR *FORM*). ADD.PROCESS may pack the name with a number to make it unique. Process names are treated as case-insensitive strings. This name is solely for the convenience of manipulating processes at Lisp type-in; e.g., the name can be given as the *PROC* argument to most process functions, and the name appears in menus of processes. However, programs should normally only deal in process handles, both for efficiency and to avoid the confusion that can result if two processes have the same defining form.
SUSPEND	If the value is non-NIL, the new process is created but then immediately suspended; i.e., the process does not actually run until woken by a WAKE.PROCESS (below).

(**PROCESSPROP** *PROC PROP NEWVALUE*)   [NoSpread Function]

Used to get or set the values of certain properties of process *PROC*, in a manner analogous to WINDOWPROP. If *NEWVALUE* is supplied (including if it is NIL), property *PROP* is given that value. In all cases, returns the old value of the property. The following properties have special meaning for processes; all others are uninterpreted:

NAME	Value is a symbol used for identifying the process to the user.
FORM	Value is the Lisp form used to start the process (readonly).
RESTARTABLE	Value is a flag indicating the disposition of the process following errors or hard resets:

NIL or NO (the default): If an untrapped error (or Control-E or Control-D) causes its form to be exited, the process is deleted. The process is also deleted if a HARDRESET (or Control-D from RAID) occurs, causing the entire Process world to be reinitialized.

T or YES: The process is automatically restarted on errors or HARDRESET. This is the normal setting for persistent "background" processes, such as the mouse process, that can safely restart themselves on errors.

HARDRESET: The process is deleted as usual if an error causes its form to be exited, but it *is* restarted on a HARDRESET. This setting is preferred for persistent processes for which an error is an unusual condition, one

that might repeat itself if the process were simply blindly restarted.

RESTARTFORM If the value is non-`NIL`, it is the form used if the process is restarted (instead of the value of the `FORM` property). Of course, the process must also have a non-`NIL` `RESTARTABLE` prop for this to have any effect.

BEFOREEXIT If the value is the atom `DON'T`, it will not be interrupted by a `LOGOUT`. If `LOGOUT` is attempted before the process finishes, a message will appear saying that Interlisp is waiting for the process to finish. If you want the `LOGOUT` to proceed without waiting, you must use the process status window (from the background menu) to delete the process.

AFTEREXIT Value indicates the disposition of the process following a resumption of Lisp after some exit (`LOGOUT`, `SYSOUT`, `MAKESYS`). Possible values are:

`DELETE`: Delete the process.

`SUSPEND`: Suspend the process; i.e., do not let it run until it is explicitly woken.

An event: Cause the process to be suspended waiting for the event (See the Events section below).

INFOHOOK Value is a function or form used to provide information about the process, in conjunction with the `INFO` command in the process status window (see the Process Status Window section below).

WINDOW Value is a window associated with the process, the process's "main" window. Used to switch the tty process to this process when you click in this window (see the Switching the TTY Process section below).

Setting the `WINDOW` property does not set the primary I/O stream (`NIL`) or the terminal I/O stream (`T`) to the window. When a process is created, I/O operations to the `NIL` or `T` stream will cause a new window to appear. `TTYDISPLAYSTREAM` (see Chapter 26) should be used to set the terminal I/O stream of a process to a specific window.

TTYENTRYFN Value is a function that is applied to the process when the process is made the tty process (see the Switching the TTY Process section below).

TTYEXITFN   Value is a function that is applied to the process when the process ceases to be the tty process (see the Switching the TTY Process section below).

(**THIS.PROCESS**)   [Function]

Returns the handle of the currently running process, or NIL if the Process world is turned off.

(**DEL.PROCESS** *PROC*)   [Function]

Deletes process *PROC*. *PROC* may be a process handle (returned by ADD.PROCESS), or its name. If *PROC* is the currently running process, DEL.PROCESS does not return!

(**PROCESS.RETURN** *VALUE*)   [Function]

Terminates the currently running process, causing it to "return" *VALUE*. There is an implicit PROCESS.RETURN around the *FORM* argument given to ADD.PROCESS, so that normally a process can finish by simply returning; PROCESS.RETURN is supplied for earlier termination.

(**PROCESS.RESULT** *PROCESS WAITFORRESULT*)   [Function]

If *PROCESS* has terminated, returns the value, if any, that it returned. This is either the value of a PROCESS.RETURN or the value returned from the form given to ADD.PROCESS. If the process was aborted, the value is NIL. If *WAITFORRESULT* is true, PROCESS.RESULT blocks until *PROCESS* finishes, if necessary; otherwise, it returns NIL immediately if *PROCESS* is still running. *PROCESS* must be the actual process handle returned from ADD.PROCESS, not a process name, as the association between handle and name disappears when the process finishes (and the process handle itself is then garbage collected if no one else has a pointer to it).

(**PROCESS.FINISHEDP** *PROCESS*)   [Function]

True if *PROCESS* has terminated. The value returned is an indication of how it finished: NORMAL or ERROR.

(**PROCESSP** *PROC*)   [Function]

True if *PROC* is the handle of an active process, i.e., one that has not yet finished.

(**RELPROCESSP** *PROCHANDLE*)   [Function]

True if *PROCHANDLE* is the handle of a deleted process. This is analogous to RELSTKP. It differs from PROCESS.FINISHEDP in that it never causes an error, while PROCESS.FINISHEDP can cause an error if its *PROC* argument is not a process at all.

(**RESTART.PROCESS** *PROC*)   [Function]

Unwinds *PROC* to its top level and reevaluates its form. This is effectively a DEL.PROCESS followed by the original ADD.PROCESS.

(**MAP.PROCESSES** *MAPFN*) [Function]

> Maps over all processes, calling *MAPFN* with three arguments: the process handle, its name, and its form.

(**FIND.PROCESS** *PROC ERRORFLG*) [Function]

> If *PROC* is a process handle or the name of a process, returns the process handle for it, else NIL. If *ERRORFLG* is T, generates an error if *PROC* is not, and does not name, a live process.

Process Control Constructs

(**BLOCK** *MSECSWAIT TIMER*) [Function]

> Yields control to the next waiting process, assuming any is ready to run. If *MSECSWAIT* is specified, it is a number of milliseconds to wait before returning, or T, meaning wait forever (until explicitly woken). Alternatively, *TIMER* can be given as a millisecond timer (as returned by SETUPTIMER, Chapter 12) of an absolute time at which to wake up. In any of those cases, the process enters the *waiting* state until the time limit is up. BLOCK with no arguments leaves the process in the *runnable* state, i.e., it returns as soon as every other runnable process of the same priority has had a chance.
>
> BLOCK can be aborted by interrupts such as Control-D, Control-E, or Control-B. BLOCK will return before its timeout is completed, if the process is woken by WAKE.PROCESS, PROCESS.EVAL, or PROCESS.APPLY.

(**DISMISS** *MSECSWAIT TIMER NOBLOCK*) [Function]

> DISMISS is used to dismiss the current process for a given period of time. Similar to BLOCK, except that:
>
> - DISMISS is guaranteed not to return until the specified time has elapsed
>
> - *MSECSWAIT* cannot be T to wait forever
>
> - If *NOBLOCK* is T, DISMISS will not allow other processes to run, but will busy-wait until the amount of time given has elapsed.

(**WAKE.PROCESS** *PROC STATUS*) [Function]

> Explicitly wakes process *PROC*, i.e., makes it *runnable*, and causes its call to BLOCK (or other waiting function) to return *STATUS*. This is one simple way to notify a process of some happening; however, note that if WAKE.PROCESS is applied to a process more than once before the process actually gets its turn to run, it sees only the latest *STATUS*.

(**SUSPEND.PROCESS** *PROC*)   [Function]

>   Blocks process *PROC* indefinitely, i.e., *PROC* will not run until it is woken by a
>   WAKE.PROCESS.

The following three functions allow access to the stack context of some other process. They require a
little bit of care, and are computationally non-trivial, but they do provide a more powerful way of
manipulating another process than WAKE.PROCESS allows.

(**PROCESS.EVALV** *PROC VAR*)   [Function]

>   Performs (EVALV *VAR*) in the stack context of *PROC*.

(**PROCESS.EVAL** *PROC FORM WAITFORRESULT*)   [Function]

>   Evaluates *FORM* in the stack context of *PROC*. If *WAITFORRESULT* is true, blocks until the
>   evaluation returns a result, else allows the current process to run in parallel with the
>   evaluation. Any errors that occur will be in the context of *PROC*, so be careful. In
>   particular, note that
>
>   (PROCESS.EVAL *PROC* '(NLSETQ (FOO)))
>
>   and
>
>   (NLSETQ (PROCESS.EVAL *PROC* '(FOO)))
>
>   behave quite differently if FOO causes an error. And it is quite permissible to intentionally
>   cause an error in proc by performing
>
>   (PROCESS.EVAL *PROC* '(ERROR!))
>
>   If *WAITFORRESULT* is true and the computation in the other process aborts or the other
>   process is killed PROCESS.EVAL returns :ABORTED
>
>   After *FORM* is evaluated in *PROC*, the process *PROC* is woken up, even if it was running
>   BLOCK or AWAIT.EVENT. This is necessary because an event of interest may have
>   occurred while the process was evaluating *FORM*.

(**PROCESS.APPLY** *PROC FN ARGS WAITFORRESULT*)   [Function]

>   Performs (APPLY *FN ARGS*) in the stack context of *PROC*. Note the same warnings as
>   with PROCESS.EVAL.

Events

An "event" is a synchronizing primitive used to coordinate related processes, typically producers and
consumers. Consumer processes can "wait" on events, and producers "notify" events.

(**CREATE.EVENT** *NAME*)   [Function]

> Returns an instance of the EVENT datatype, to be used as the event argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(**AWAIT.EVENT** *EVENT TIMEOUT TIMERP*)   [Function]

> Suspends the current process until *EVENT* is notified, or until a timeout occurs. If *TIMEOUT* is NIL, there is no timeout. Otherwise, timeout is either a number of milliseconds to wait, or, if *TIMERP* is T, a millisecond timer set to expire at the desired time using SETUPTIMER (see Chapter 12).

(**NOTIFY.EVENT** *EVENT ONCEONLY*)   [Function]

> If there are processes waiting for *EVENT* to occur, causes those processes to be placed in the running state, with *EVENT* returned as the value from AWAIT.EVENT. If *ONCEONLY* is true, only runs the first process waiting for the event (this should only be done if the programmer knows that there can only be one process capable of responding to the event at once).

The meaning of an event is up to the programmer. In general, however, the notification of an event is merely a hint that something of interest to the waiting process has happened; the process should still verify that the conceptual event actually occurred. That is, *the process should be written so that it operates correctly even if woken up before the timeout and in the absence of the notified event.* In particular, the completion of PROCESS.EVAL and related operations in effect wakes up the process in which they were performed, since there is no secure way of knowing whether the event of interest occurred while the process was busy performing the PROCESS.EVAL.

There is currently one class of system-defined events, used with the network code. Each Pup and NS socket has associated with it an event that is notified when a packet arrives on the socket; the event can be obtained by calling PUPSOCKETEVENT or NSOCKETEVENT, respectively (see Chapter 30).

Monitors

It is often the case that cooperating processes perform operations on shared structures, and some mechanism is needed to prevent more than one process from altering the structure at the same time. Some languages have a construct called a monitor, a collection of functions that access a common structure with mutual exclusion provided and enforced by the compiler via the use of monitor locks. Medley has taken this implementation notion as the basis for a mutual exclusion capability suitable for a dynamically-scoped environment.

A monitorlock is an object created by you and associated with (e.g., stored in) some shared structure that is to be protected from simultaneous access. To access the structure, a program waits for the lock to be free, then takes ownership of the lock, accesses the structure, then releases the lock. The functions and macros below are used:

(**CREATE.MONITORLOCK** *NAME*) [Function]

> Returns an instance of the MONITORLOCK datatype, to be used as the lock argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(**WITH.MONITOR** *LOCK FORM$_1$ ... FORM$_N$*) [Macro]

> Evaluates *FORM$_1$ ... FORM$_N$* while owning *LOCK*, and returns the value of *FORM$_N$*. This construct is implemented so that the lock is released even if the form is exited via error (currently implemented with RESETLST).

> Ownership of a lock is dynamically scoped: if the current process already owns the lock (e.g., if the caller was itself inside a WITH.MONITOR for this lock), WITH.MONITOR does not wait for the lock to be free before evaluating *FORM$_1$ ... FORM$_N$*.

(**WITH.FAST.MONITOR** *LOCK FORM$_1$ ... FORM$_N$*) [Macro]

> Like WITH.MONITOR, but implemented without the RESETLST. User interrupts (e.g., Control-E) are inhibited during the evaluation of *FORM$_1$ ... FORM$_N$*.

> Programming restriction: the evaluation of *FORM$_1$ ... FORM$_N$* must not error (the lock would not be released). This construct is mainly useful when the forms perform a small, safe computation that never errors and need never be interrupted.

(**MONITOR.AWAIT.EVENT** *RELEASELOCK EVENT TIMEOUT TIMERP*) [Function]

> For use in blocking inside a monitor. Performs (AWAIT.EVENT *EVENT TIMEOUT TIMERP*), but releases *RELEASELOCK* first, and reobtains the lock (possibly waiting) on wakeup.

> Typical use for MONITOR.AWAIT.EVENT: A function wants to perform some operation on *FOO*, but only if it is in a certain state. It has to obtain the lock on the structure to make sure that the state of the structure does not change between the time it tests the state and performs the operation. If the state turns out to be bad, it then waits for some other process to make the state good, meanwhile releasing the lock so that the other process can alter the structure.

```
(WITH.MONITOR FOO-LOCK
 (until CONDITION-OF-FOO
  do (MONITOR.AWAIT.EVENT FOO-LOCK EVENT-FOO-
CHANGED TIMEOUT))
    OPERATE-ON-FOO)
```

> It is sometimes convenient for a process to have WITH.MONITOR at its top level and then do all its interesting waiting using MONITOR.AWAIT.EVENT. Not only is this often cleaner, but in the present implementation in cases where the lock is frequently accessed, it saves the RESETLST overhead of WITH.MONITOR.

> Programming restriction: There must not be an ERRORSET between the enclosing WITH.MONITOR and the call to MONITOR.AWAIT.EVENT such that the ERRORSET would catch an ERROR! and continue inside the monitor, for the lock would not have been

reobtained. (The reason for this restriction is that, although MONITOR.AWAIT.EVENT won't itself error, you could have caused an error with an interrupt, or a PROCESS.EVAL in the context of the waiting process that produced an error.)

On rare occasions it may be useful to manipulate monitor locks directly. The following two functions are used in the implementation of WITH.MONITOR:

(**OBTAIN.MONITORLOCK** *LOCK DONTWAIT UNWINDSAVE*) [Function]

Takes possession of *LOCK*, waiting if necessary until it is free, unless *DONTWAIT* is true, in which case it returns NIL immediately. If *UNWINDSAVE* is true, performs a RESETSAVE to be unwound when the enclosing RESETLST exits. Returns *LOCK* if *LOCK* was successfully obtained, T if the current process already owned *LOCK*.

(**RELEASE.MONITORLOCK** *LOCK EVENIFNOTMINE*) [Function]

Releases *LOCK* if it is owned by the current process, and wakes up the next process, if any, waiting to obtain the lock.

If *EVENIFNOTMINE* is non-NIL, the lock is released even if it is not owned by the current process.

When a process is deleted, any locks it owns are released.

Global Resources

The biggest source of problems in the multi-processing environment is the matter of global resources. Two processes cannot both use the same global resource if there can be a process switch in the middle of their use (currently this means calls to BLOCK, but ultimately with a preemptive scheduler means anytime). Thus, user code should be wary of its own use of global variables, if it ever makes sense for the code to be run in more than one process at a time. "State" variables private to a process should generally be bound in that process; structures that are shared among processes (or resources used privately but expensive to duplicate per process) should be protected with monitor locks or some other form of synchronization.

Aside from user code, however, there are many *system* global variables and resources. Most of these arise historically from the single-process Interlisp-10 environment, and will eventually be changed in Medley to behave appropriately in a multi-processing environment. Some have already been changed, and are described below. Two other resources not generally thought of as global variables— the keyboard and the mouse—are particularly idosyncratic, and are discussed in the next section.

are allocated per process in Medley: primary input and output (the streams affected by INPUT and OUTPUT), terminal input and output (the streams designated by the name T), the primary read table and primary terminal table, and dribble files. Thus, each process can print to its own primary output,

print to the terminal, read from a different primary input, all without interfering with another process's reading and printing.

Each process begins life with its primary and terminal input/output streams set to a dummy stream. If the process attempts input or output using any of those dummy streams, e.g., by calling (READ T), or (PRINT & T), a tty window is automatically created for the process, and that window becomes the primary input/output and terminal input/output for the process. The default tty window is created at or near the region specified in the variable DEFAULTTTYREGION.

A process can, of course, call TTYDISPLAYSTREAM explicitly to give itself a tty window of its own choosing, in which case the automatic mechanism never comes into play. Calling TTYDISPLAYSTREAM when a process has no tty window not only sets the terminal streams, but also sets the primary input and output streams to be that window, assuming they were still set to the dummy streams.

> (**HASTTYWINDOWP** *PROCESS*)   [Function]
>
>> Returns T if the process *PROCESS* has a tty window; NIL otherwise. If *PROCESS* is NIL, it defaults to the current process.

Other system resources that are typically changed by RESETFORM, RESETLST, or RESETVARS are all global entities. In the multiprocessing environment, these constructs are suspect, as there is no provision for "undoing" them when a process switch occurs. For example, in the current release of Medley, it is not possible to set the print radix to 8 inside only one process, as the print radix is a global entity.

Note that RESETFORM and similar expressions are perfectly valid in the process world, and even quite useful, when they manipulate things strictly within one process. The process world is arranged so that deleting a process also unwinds any RESETxxx expressions that were performed in the process and are still waiting to be unwound, exactly as if a Control-D had reset the process to the top. Additionally, there is an implicit RESETLST at the top of each process, so that RESETSAVE can be used as a way of providing "cleanup" functions for when a process is deleted. For these, the value of RESETSTATE (see Chapter 14) is NIL if the process finished normally, ERROR if it was aborted by an error, RESET if the process was explicitly deleted, and HARDRESET if the process is being restarted after a HARDRESET or a RESTART.PROCESS.

Typein and the TTY Process

There is one global resource, the keyboard, that is particularly problematic to share among processes. Consider, for example, having two processes both performing (READ T). Since the keyboard input routines block while there is no input, both processes would spend most of their time blocking, and it would simply be a matter of chance which process received each character of type-in.

To resolve such dilemmas, the system designates a distinguished process, termed the *tty process*, that is assumed to be the process that is involved in terminal interaction. Any type-in from the keyboard goes to that process. If a process other than the tty process requests keyboard input, it blocks until it becomes the tty process. When the tty process is switched (in any of the ways described further below), any typeahead that occurred before the switch is saved and associated with the current tty process. Thus, it is always the case that keystrokes are sent to the process that is the tty process at the time of the keystrokes, regardless of when that process actually gets around to reading them.

> **BACKGROUNDFNS** [Variable]
>
> > A list of functions to call "in the background". The system runs a process (called "BACKGROUND") whose sole task is to call each of the functions on the list BACKGROUNDFNS repeatedly. Each element is the name of a function of no arguments. This is a good place to put cheap background tasks that only do something once in a while and hence do not want to spend their own separate process on it. However, note that it is considered good citizenship for a background function with a time-consuming task to spawn a separate process to do it, so that the other background functions are not delayed.

> **TTYBACKGROUNDFNS** [Variable]
>
> > This list is like BACKGROUNDFNS, but the functions are only called while in a tty input wait. That is, they always run in the tty process, and only when the user is not actively typing. For example, the flashing caret is implemented by a function on this list. Again, functions on this list should spend very little time (much less than a second), or else spawn a separate process.

It is less immediately obvious how to handle keyboard interrupt characters, as their action is asynchronous and not always tied to type-in. Interrupt handling is described in the Handling of Interrupts section below.

Switching the TTY Process

Any process can make itself be the tty process by calling `TTY.PROCESS`.

> (**TTY.PROCESS** *PROC*) [Function]
>
> > Returns the handle of the current tty process. In addition, if *PROC* is non-`NIL`, makes it be the tty process. The special case of *PROC* = `T` is interpreted to mean the executive process; this is sometimes useful when a process wants to explicitly give up being the tty process.

> (**TTY.PROCESSP** *PROC*) [Function]
>
> > True if *PROC* is the tty process; *PROC* defaults to the running process. Thus, (`TTY.PROCESSP`) is true if the caller is the tty process.

(**WAIT.FOR.TTY** *MSECS NEEDWINDOW*)   [Function]

> Efficiently waits until (TTY.PROCESSP) is true. WAIT.FOR.TTY is called internally by the system functions that read from the terminal; user code thus need only call it in special cases.

> If *MSECS* is non-NIL, it is the number of milliseconds to wait before timing out. If WAIT.FOR.TTY times out before (TTY.PROCESSP) is true, it returns NIL, otherwise it returns T. If *MSECS* is NIL, WAIT.FOR.TTY will not time out.

> If *NEEDWINDOW* is non-NIL, WAIT.FOR.TTY opens a TTY window for the current process if one isn't already open.

> WAIT.FOR.TTY spawns a new mouse process if called under the mouse process (see SPAWN.MOUSE, in the Keeping the Mouse Alive section below).

In some cases, such as in functions invoked as a result of mouse action or a user's typed-in call, it is reasonable for the function to invoke TTY.PROCESS itself so that it can take subsequent user type in. In other cases, however, this is too undisciplined; it is desirable to let you designate which process type-in should be directed to. This is most conveniently done by mouse action.

The system supports the model that "to type to a process, you click in its window." To cooperate with this model, any process desiring keyboard input should put its process handle as the PROCESS property of its window(s). To handle the common case, the function TTYDISPLAYSTREAM does this automatically when the ttydisplaystream is switched to a new window. A process can own any number of windows; clicking in any of those windows gives the process the tty.

This mechanism suffices for most casual process writers. For example, if a process wants all its input/output interaction to occur in a particular window that it has created, it should just make that window be its tty window by calling TTYDISPLAYSTREAM. Thereafter, it can PRINT or READ to/from the T stream; if the process is not the tty process at the time that it calls READ, it will block until the user clicks in the window.

For those needing tighter control over the tty, the default behavior can be overridden or supplemented. The remainder of this section describes the mechanisms involved.

There is a window property WINDOWENTRYFN that controls whether and how to switch the tty to the process owning a window. The mouse handler, before invoking any normal BUTTONEVENTFN, specifically notices the case of a button going down in a window that belongs to a process (i.e., has a PROCESS window property) that is not the tty process. In this case, it invokes the window's WINDOWENTRYFN of one argument (WINDOW). WINDOWENTRYFN defaults to GIVE.TTY.PROCESS:

(**GIVE.TTY.PROCESS** *WINDOW*)   [Function]

> If *WINDOW* has a PROCESS property, performs (TTY.PROCESS (WINDOWPROP *WINDOW* 'PROCESS)) and then invokes *WINDOW*'s BUTTONEVENTFN function (or RIGHTBUTTONFN if the right button is down).

There are some cases where clicking in a window does not always imply that the user wants to talk to that window. For example, clicking in a text editor window with a shift key held down means to "shift-select" some piece of text into the input buffer of the *current* tty process. The editor supports this by supplying a WINDOWENTRYFN that performs GIVE.TTY.PROCESS if no shift key is down, but goes into its shift-select mode, without changing the tty process, if a shift key is down. The shift-select mode performs a BKSYSBUF of the selected text when the shift key is let up, the BKSYSBUF feeding input to the current tty process.

Sometimes a process wants to be notified when it becomes the tty process, or stops being the tty process. To support this, there are two process properties, TTYEXITFN and TTYENTRYFN. The actions taken by TTY.PROCESS when it switches the tty to a new process are as follows: the former tty process's TTYEXITFN is called with two arguments (*OLDTTYPROCESS NEWTTYPROCESS*); the new process is made the tty process; finally, the new tty process's TTYENTRYFN is called with two arguments (*NEWTTYPROCESS OLDTTYPROCESS*). Normally the TTYENTRYFN and TTYEXITFN need only their first argument, but the other process involved in the switch is supplied for completeness. In the present system, most processes want to interpret the keyboard in the same way, so it is considered the responsibility of any process that changes the keyboard interpretation to restore it to the normal state by its TTYEXITFN.

A window is "owned" by the last process that anyone gave as the window's PROCESS property. Ordinarily there is no conflict here, as processes tend to own disjoint sets of windows (though, of course, cooperating processes can certainly try to confuse each other). The only likely problem arises with that most global of windows, PROMPTWINDOW. Programs should not be tempted to read from PROMPTWINDOW. This is not usually necessary anyway, as the first attempt to read from T in a process that has not set its TTYDISPLAYSTREAM to its own window causes a tty window to be created for the process (see the Global Resources section above).

Handling of Interrupts

At the time that a keyboard interrupt character (see Chapter 29) is struck, any process could be running, and some decision must be made as to which process to actually interrupt. To the extent that keyboard interrupts are related to type-in, most interrupts are taken in the tty process; however, the following are handled specially:

RESET (initially Control-D)
ERROR (initially Control-E)   These interrupts are taken in the mouse process, if the mouse is not in its idle state; otherwise they are taken in the tty process. Thus, Control-E can be used to abort some mouse-invoked window action, such as the Shape command. As a consequence, note that if the mouse invokes some lengthy computation that the user thinks of as "background", Control-E still aborts it, even though that may not have been what the user intended. Such lengthy computations, for various reasons, should generally be

performed by spawning a separate process to perform them. The `RESET` interrupt in a process other than the executive is interpreted exactly as if an error unwound the process to its top level: if the process was designated `RESTARTABLE = T`, it is restarted; otherwise it is killed.

HELP (initially Control-G)   A menu of processes is presented to the user, who is asked to select which one the interrupt should occur in. The current tty process appears with a * next to its name at the top of the menu. The menu also includes an entry "[Spawn Mouse]", for the common case of needing a mouse because the mouse process is currently tied up running someone's `BUTTONEVENTFN`; selecting this entry spawns a new mouse process, and no break occurs.

BREAK (initially Control-B)   Performs the `HELP` interrupt in the mouse process, if the mouse is not in its idle state; otherwise it is performed in the tty process.

RUBOUT (initially `DELETE`)   This interrupt clears typeahead in *all* processes.

RAID, STACK OVERFLOW
      STORAGE FULL   These interrupts always occur in whatever process was running at the time the interrupt struck. In the cases of `STACK OVERFLOW` and `STORAGE FULL`, this means that the interrupt is more likely to strike in the offending process (especially if it is a "runaway" process that is not blocking). Note, however, that this process is still not necessarily the guilty party; it could be an innocent bystander that just happened to use up the last of a resource prodigiously consumed by some other process.

Keeping the Mouse Alive

Since the window mouse handler runs in its own process, it is not available while a window's `BUTTONEVENTFN` function (or any of the other window functions invoked by mouse action) is running. This leads to two sorts of problems: (1) a long computation underneath a `BUTTONEVENTFN` deprives the user of the mouse for other purposes, and (2) code that runs as a `BUTTONEVENTFN`

cannot rely on other BUTTONEVENTFNs running, which means that there some pieces of code that run differently from normal when run under the mouse process. These problems are addressed by the following functions:

(**SPAWN.MOUSE** —)   [Function]

> Spawns another mouse process, allowing the mouse to run even if it is currently "tied up" under the current mouse process. This function is intended mainly to be typed in at the Lisp executive when you notice the mouse is busy.

(**ALLOW.BUTTON.EVENTS**)   [Function]

> Performs a (SPAWN.MOUSE) only when called underneath the mouse process. This should be called (once, on entry) by any function that relies on BUTTONEVENTFNs for completion, if there is any possibility that the function will itself be invoked by a mouse function.

It never hurts, at least logically, to call SPAWN.MOUSE or ALLOW.BUTTON.EVENTS needlessly, as the mouse process arranges to quietly kill itself if it returns from the user's BUTTONEVENTFN and finds that another mouse process has sprung up in the meantime. (There is, of course, some computational expense.)

Process Status Window

The background menu command PSW (see Chapter 27) and the function PROCESS.STATUS.WINDOW (below) create a "Process Status Window", that allows you to examine and manipulate all of the existing processes:

```
SPACEWINDOW
   TEdit
   MOUSE
  ERIS#LEAF
\10MBWATCHER
    EXEC
\NSGATELISTENER
\PUPGATELISTENER
\TIMER.PROCESS
 BACKGROUND
----------------------------
 BT      WHO?     KILL
 BTV     KBD←   RESTART
 BTV*    INFO     WAKE
 BTV!   BREAK   SUSPEND
```

The window consists of two menus. The top menu lists all the processes at the moment. Commands in the bottom menu operate on the process selected in the top menu (EXEC in the example above). The commands are:

> **BT**, **BTV**, **BTV***, **BTV!**   Displays a backtrace of the selected process.

WHO?	Changes the selection to the tty process, i.e., the one currently in control of the keyboard.
KBD←	Associates the keyboard with the selected process; i.e., makes the selected process be the tty process.
INFO	If the selected process has an INFOHOOK property, calls it. The hook may be a function, which is then applied to two arguments, the process and the button (LEFT or MIDDLE) used to invoke INFO, or a form, which is simply EVAL'ed. The APPLY or EVAL happens in the context of the selected process, using PROCESS.APPLY or PROCESS.EVAL. The INFOHOOK process property can be set using PROCESSPROP (see the Creating and Destroying Processes section above).
BREAK	Enter a break under the selected process. This has the side effect of waking the process with the value returned from the break.
KILL	Deletes the selected process.
RESTART	Restarts the selected process.
WAKE	Wakes the selected process. Prompts for a value to wake it with (see WAKE.PROCESS).
SUSPEND	Suspends the selected process; i.e., causes it to block indefinitely (until explicitly woken).

(**PROCESS.STATUS.WINDOW** *WHERE*)   [Function]

Puts up a process status window that provides several debugging commands for manipulating running processes. If the window is already up, PROCESS.STATUS.WINDOW refreshes it. If *WHERE* is a position, the window is placed in that position; otherwise, you are prompted for a position.

Currently, the process status window runs under the mouse process, like other menus, so if the mouse is unavailable (e.g., a mouse function is performing an extensive computation), you may be unable to use the process status window (you can try SPAWN.MOUSE, of course).

Non-Process Compatibility

This section describes some considerations for authors of programs that ran in the old single-process Medley environment, and now want to make sure they run properly in the multi-processing world. The biggest problem to watch out for is code that runs underneath the mouse handler. Writers of mouse handler functions should remember that in the process world the mouse handler runs in its own process, and hence (a) you cannot depend on finding information on the stack (stash it in the

window instead), and (b) while your function is running, the mouse is not available (if you have any non-trivial computation to do, spawn a process to do it, notify one of your existing processes to do it, or use PROCESS.EVAL to run it under some other process).

The following functions are meaningful even if the process world is not on: BLOCK (invokes the system background routine, which includes handling the mouse); TTY.PROCESS, THIS.PROCESS (both return NIL); and TTY.PROCESSP (returns T, i.e., anyone is allowed to take tty input). In addition, the following two functions exist in both worlds:

(**EVAL.AS.PROCESS** *FORM*) [Function]

Same as (ADD.PROCESS *FORM* 'RESTARTABLE 'NO), when processes are running, EVAL when not. This is highly recommended for mouse functions that perform any non-trivial activity.

(**EVAL.IN.TTY.PROCESS** *FORM WAITFORRESULT*) [Function]

Same as (PROCESS.EVAL (TTY.PROCESS) *FORM WAITFORRESULT*), when processes are running, EVAL when not.

Most of the process functions that do not take a process argument can be called even if processes aren't running. ADD.PROCESS creates, but does not run, a new process (it runs when PROCESSWORLD is called).

23. PROCESSES

The Interlisp-D Process mechanism provides an environment in which multiple Lisp processes can run in parallel. Each executes in its own stack space, but all share a global address space. The current process implementation is cooperative; i.e., process switches happen voluntarily, either when the process in control has nothing to do or when it is in a convenient place to pause. There is no preemption or guaranteed service, so you cannot run something demanding (e.g., Chat) at the same time as something that runs for long periods without yielding control. Keyboard input and network operations block with great frequency, so processes currently work best for highly interactive tasks (editing, making remote files).

In Interlisp-D, the process mechanism is already turned on, and is expected to stay on during normal operations, as some system facilities (in particular, most network operations) require it. However, under exceptional conditions, the following function can be used to turn the world off and on:

(PROCESSWORLD *FLG*)   [Function]

> Starts up the process world, or if *FLG* = OFF, kills all processes and turns it off. Normally does not return. The environment starts out with two processes: a top-level EVALQT (the initial "tty" process) and the "background" process, which runs the window mouse handler and other system background tasks.
>
> PROCESSWORLD is intended to be called at the top level of Interlisp, not from within a program. It does not toggle some sort of switch; rather, it constructs some new processes in a new part of the stack, leaving any callers of PROCESSWORLD in a now inaccessible part of the stack. Calling (PROCESSWORLD 'OFF) is the only way the call to PROCESSWORLD ever returns.

(HARDRESET)   [Function]

> Resets the whole world, and rebuilds the stack from scratch. This is "harder" than doing RESET to every process, because it also resets system internal processes (such as the keyboard handler).
>
> HARDRESET automatically turns the process world on (or resets it if it was on), unless the variable AUTOPROCESSFLG is NIL.

Creating and Destroying Processes

(ADD.PROCESS *FORM PROP$_1$ VALUE$_1$ ... PROP$_N$ VALUE$_N$*)   [NoSpread Function]

Creates a new process evaluating *FORM*, and returns its process handle. The process's stack environment is the top level, i.e., the new process does not have access to the environment in which ADD.PROCESS was called; all such information must be passed as arguments in *FORM*. The process runs until *FORM* returns or the process is explicitly deleted. An untrapped error within the process also deletes the process (unless its RESTARTABLE property is T), in which case a message is printed to that effect.

The remaining arguments are alternately property names and values. Any property/value pairs acceptable to PROCESSPROP may be given, but the following two are directly relevant to ADD.PROCESS:

    NAME   Value should be a litatom; if not given, the process name is taken from (CAR *FORM*). ADD.PROCESS may pack the name with a number to make it unique. This name is solely for the convenience of manipulating processes at Lisp typein; e.g., the name can be given as the *PROC* argument to most process functions, and the name appears in menus of processes. However, programs should normally only deal in process handles, both for efficiency and to avoid the confusion that can result if two processes have the same defining form.

    SUSPEND   If the value is non-NIL, the new process is created but then immediately suspended; i.e., the process does not actually run until woken by a WAKE.PROCESS (below).

(PROCESSPROP *PROC PROP NEWVALUE*)   [NoSpread Function]

Used to get or set the values of certain properties of process *PROC*, in a manner analogous to WINDOWPROP. If *NEWVALUE* is supplied (including if it is NIL), property *PROP* is given that value. In all cases, returns the old value of the property. The following properties have special meaning for processes; all others are uninterpreted:

    NAME   Value is a litatom used for identifying the process to the user.

    FORM   Value is the Lisp form used to start the process (readonly).

    RESTARTABLE   Value is a flag indicating the disposition of the process following errors or hard resets:

NIL or NO (the default): If an untrapped error (or Control-E or Control-D) causes its form to be exited, the process is deleted. The process is also deleted if a HARDRESET (or Control-D from RAID) occurs, causing the entire Process world to be reinitialized.

T or YES: The process is automatically restarted on errors or HARDRESET. This is the normal setting for persistent "background" processes, such as the mouse process, that can safely restart themselves on errors.

HARDRESET: The process is deleted as usual if an error causes its form to be exited, but it *is* restarted on a HARDRESET. This setting is preferred for persistent processes for which an error is an unusual condition, one that might repeat itself if the process were simply blindly restarted.

RESTARTFORM   If the value is non-NIL, it is the form used if the process is restarted (instead of the value of the FORM property). Of course, the process must also have a non-NIL RESTARTABLE prop for this to have any effect.

BEFOREEXIT   If the value is the atom DON'T, it will not be interrupted by a LOGOUT. If LOGOUT is attempted before the process finishes, a message will appear saying that Interlisp is waiting for the process to finish. If you want the LOGOUT to proceed without waiting, you must use the process status window (from the background menu) to delete the process.

AFTEREXIT   Value indicates the disposition of the process following a resumption of Lisp after some exit (LOGOUT, SYSOUT, MAKESYS). Possible values are:

DELETE: Delete the process.

SUSPEND: Suspend the process; i.e., do not let it run until it is explicitly woken.

An event: Cause the process to be suspended waiting for the event (See the Events section below).

INFOHOOK   Value is a function or form used to provide information about the process, in conjunction with the INFO command in the process status window (see the Process Status Window section below).

WINDOW	Value is a window associated with the process, the process's "main" window. Used to switch the tty process to this process when you click in this window (see the Switching the TTY Process section below).
	Setting the WINDOW property does not set the primary I/O stream (NIL) or the terminal I/O stream (T) to the window. When a process is created, I/O operations to the NIL or T stream will cause a new window to appear. TTYDISPLAYSTREAM (see Chapter 28) should be used to set the terminal i/o stream of a process to a specific window.
TTYENTRYFN	Value is a function that is applied to the process when the process is made the tty process (see the Switching the TTY Process section below).
TTYEXITFN	Value is a function that is applied to the process when the process ceases to be the tty process (see the Switching the TTY Process section below).

(THIS.PROCESS) [Function]

Returns the handle of the currently running process, or NIL if the Process world is turned off.

(DEL.PROCESS *PROC* —) [Function]

Deletes process *PROC*. *PROC* may be a process handle (returned by ADD.PROCESS), or its name. If *PROC* is the currently running process, DEL.PROCESS does not return!

(PROCESS.RETURN *VALUE*) [Function]

Terminates the currently running process, causing it to "return" *VALUE*. There is an implicit PROCESS.RETURN around the *FORM* argument given to ADD.PROCESS, so that normally a process can finish by simply returning; PROCESS.RETURN is supplied for earlier termination.

(PROCESS.RESULT *PROCESS WAITFORRESULT*) [Function]

If *PROCESS* has terminated, returns the value, if any, that it returned. This is either the value of a PROCESS.RETURN or the value returned from the form given to ADD.PROCESS. If the process was aborted, the value is NIL. If *WAITFORRESULT* is true, PROCESS.RESULT blocks until *PROCESS* finishes, if necessary; otherwise, it returns NIL immediately if *PROCESS* is still running. *PROCESS* must be the actual process handle returned from

ADD.PROCESS, not a process name, as the association between handle and name disappears when the process finishes (and the process handle itself is then garbage collected if no one else has a pointer to it).

(PROCESS.FINISHEDP *PROCESS*)   [Function]

   True if *PROCESS* has terminated. The value returned is an indication of how it finished: NORMAL or ERROR.

(PROCESSP *PROC*)   [Function]

   True if *PROC* is the handle of an active process, i.e., one that has not yet finished.

(RELPROCESSP *PROCHANDLE*)   [Function]

   True if *PROCHANDLE* is the handle of a deleted process. This is analogous to RELSTKP. It differs from PROCESS.FINISHEDP in that it never causes an error, while PROCESS.FINISHEDP can cause an error if its *PROC* argument is not a process at all.

(RESTART.PROCESS *PROC*)   [Function]

   Unwinds *PROC* to its top level and reevaluates its form. This is effectively a DEL.PROCESS followed by the original ADD.PROCESS.

(MAP.PROCESSES *MAPFN*)   [Function]

   Maps over all processes, calling *MAPFN* with three arguments: the process handle, its name, and its form.

(FIND.PROCESS *PROC ERRORFLG*)   [Function]

   If *PROC* is a process handle or the name of a process, returns the process handle for it, else NIL. If *ERRORFLG* is T, generates an error if *PROC* is not, and does not name, a live process.

Process Control Constructs

(BLOCK *MSECSWAIT TIMER*)   [Function]

   Yields control to the next waiting process, assuming any is ready to run. If *MSECSWAIT* is specified, it is a number of milliseconds to wait before returning, or T, meaning wait forever (until explicitly woken). Alternatively, *TIMER* can be given as a millisecond timer (as returned by SETUPTIMER, Chapter 12) of an absolute time at which to wake up. In any of those cases, the

process enters the *waiting* state until the time limit is up. `BLOCK` with no arguments leaves the process in the *runnable* state, i.e., it returns as soon as every other runnable process of the same priority has had a chance.

`BLOCK` can be aborted by interrupts such as Control-D, Control-E, or Control-B. `BLOCK` will return before its timeout is completed, if the process is woken by `WAKE.PROCESS`, `PROCESS.EVAL`, or `PROCESS.APPLY`.

(`DISMISS` *MSECSWAIT TIMER NOBLOCK*)   [Function]

`DISMISS` is used to dismiss the current process for a given period of time. Similar to `BLOCK`, except that:

- `DISMISS` is guaranteed not to return until the specified time has elapsed

- *MSECSWAIT* cannot be `T` to wait forever

- If *NOBLOCK* is `T`, `DISMISS` will not allow other processes to run, but will busy-wait until the amount of time given has elapsed.

(`WAKE.PROCESS` *PROC STATUS*)   [Function]

Explicitly wakes process *PROC*, i.e., makes it *runnable*, and causes its call to `BLOCK` (or other waiting function) to return *STATUS*. This is one simple way to notify a process of some happening; however, note that if `WAKE.PROCESS` is applied to a process more than once before the process actually gets its turn to run, it sees only the latest *STATUS*.

(`SUSPEND.PROCESS` *PROC*)   [Function]

Blocks process *PROC* indefinitely, i.e., *PROC* will not run until it is woken by a `WAKE.PROCESS`.

The following three functions allow access to the stack context of some other process. They require a little bit of care, and are computationally non-trivial, but they do provide a more powerful way of manipulating another process than `WAKE.PROCESS` allows.

(`PROCESS.EVALV` *PROC VAR*)   [Function]

Performs (`EVALV` *VAR*) in the stack context of *PROC*.

(`PROCESS.EVAL` *PROC FORM WAITFORRESULT*)   [Function]

Evaluates *FORM* in the stack context of *PROC*. If *WAITFORRESULT* is true, blocks until the evaluation returns a result, else allows the current process to run in parallel with the evaluation. Any errors that occur will be in the context of *PROC*, so be careful. In particular, note that

        (`PROCESS.EVAL` *PROC* '(NLSETQ (FOO)))

and

```
(NLSETQ (PROCESS.EVAL PROC '(FOO)))
```

behave quite differently if FOO causes an error. And it is quite permissible to
intentionally cause an error in proc by performing

```
(PROCESS.EVAL PROC '(ERROR!))
```

If errors are possible and *WAITFORRESULT* is true, the caller should almost
certainly make sure that *FORM* traps the errors; otherwise the caller could end
up waiting forever if *FORM* unwinds back into the pre-existing stack context of
PROC.

After *FORM* is evaluated in *PROC*, the process *PROC* is woken up, even if it
was running BLOCK or AWAIT.EVENT. This is necessary because an event of
interest may have occurred while the process was evaluating *FORM.*

(PROCESS.APPLY *PROC FN ARGS WAITFORRESULT*)   [Function]

Performs (APPLY *FN ARGS*) in the stack context of *PROC*. Note the same
warnings as with PROCESS.EVAL.

Events

An "event" is a synchronizing primitive used to coordinate related processes, typically
producers and consumers. Consumer processes can "wait" on events, and producers
"notify" events.

(CREATE.EVENT *NAME*)   [Function]

Returns an instance of the EVENT datatype, to be used as the event argument to
functions listed below. *NAME* is arbitrary, and is used for debugging or status
information.

(AWAIT.EVENT *EVENT TIMEOUT TIMERP*)   [Function]

Suspends the current process until *EVENT* is notified, or until a timeout occurs.
If *TIMEOUT* is NIL, there is no timeout. Otherwise, timeout is either a number
of milliseconds to wait, or, if *TIMERP* is T, a millisecond timer set to expire at
the desired time using SETUPTIMER (see Chapter 12).

(NOTIFY.EVENT *EVENT ONCEONLY*)   [Function]

If there are processes waiting for *EVENT* to occur, causes those processes to be
placed in the running state, with *EVENT* returned as the value from
AWAIT.EVENT. If *ONCEONLY* is true, only runs the first process waiting for

the event (this should only be done if the programmer knows that there can only be one process capable of responding to the event at once).

The meaning of an event is up to the programmer. In general, however, the notification of an event is merely a hint that something of interest to the waiting process has happened; the process should still verify that the conceptual event actually occurred. That is, *the process should be written so that it operates correctly even if woken up before the timeout and in the absence of the notified event.* In particular, the completion of `PROCESS.EVAL` and related operations in effect wakes up the process in which they were performed, since there is no secure way of knowing whether the event of interest occurred while the process was busy performing the `PROCESS.EVAL`.

There is currently one class of system-defined events, used with the network code. Each Pup and NS socket has associated with it an event that is notified when a packet arrives on the socket; the event can be obtained by calling `PUPSOCKETEVENT` or `NSOCKETEVENT`, respectively (see Chapter 32).

Monitors

It is often the case that cooperating processes perform operations on shared structures, and some mechanism is needed to prevent more than one process from altering the structure at the same time. Some languages have a construct called a monitor, a collection of functions that access a common structure with mutual exclusion provided and enforced by the compiler via the use of monitor locks. Interlisp-D has taken this implementation notion as the basis for a mutual exclusion capability suitable for a dynamically-scoped environment.

A monitorlock is an object created by you and associated with (e.g., stored in) some shared structure that is to be protected from simultaneous access. To access the structure, a program waits for the lock to be free, then takes ownership of the lock, accesses the structure, then releases the lock. The functions and macros below are used:

(`CREATE.MONITORLOCK` *NAME* —)   [Function]

    Returns an instance of the `MONITORLOCK` datatype, to be used as the lock argument to functions listed below. *NAME* is arbitrary, and is used for debugging or status information.

(`WITH.MONITOR` *LOCK FORM$_1$ ... FORM$_N$*)   [Macro]

    Evaluates *FORM$_1$ ... FORM$_N$* while owning *LOCK*, and returns the value of *FORM$_N$*. This construct is implemented so that the lock is released even if the form is exited via error (currently implemented with `RESETLST`).

Ownership of a lock is dynamically scoped: if the current process already owns the lock (e.g., if the caller was itself inside a `WITH.MONITOR` for this lock), `WITH.MONITOR` does not wait for the lock to be free before evaluating $FORM_1$ ... $FORM_N$.

(`WITH.FAST.MONITOR` *LOCK FORM$_1$ ... FORM$_N$*) [Macro]

Like `WITH.MONITOR`, but implemented without the `RESETLST`. User interrupts (e.g., Control-E) are inhibited during the evaluation of $FORM_1$ ... $FORM_N$.

Programming restriction: the evaluation of $FORM_1$ ... $FORM_N$ must not error (the lock would not be released). This construct is mainly useful when the forms perform a small, safe computation that never errors and need never be interrupted.

(`MONITOR.AWAIT.EVENT` *RELEASELOCK EVENT TIMEOUT TIMERP*) [Function]

For use in blocking inside a monitor. Performs (`AWAIT.EVENT` *EVENT TIMEOUT TIMERP*), but releases *RELEASELOCK* first, and reobtains the lock (possibly waiting) on wakeup.

Typical use for `MONITOR.AWAIT.EVENT`: A function wants to perform some operation on *FOO*, but only if it is in a certain state. It has to obtain the lock on the structure to make sure that the state of the structure does not change between the time it tests the state and performs the operation. If the state turns out to be bad, it then waits for some other process to make the state good, meanwhile releasing the lock so that the other process can alter the structure.

```
(WITH.MONITOR FOO-LOCK
  (until CONDITION-OF-FOO
  do (MONITOR.AWAIT.EVENT FOO-LOCK EVENT-FOO-
CHANGED TIMEOUT))
    OPERATE-ON-FOO)
```

It is sometimes convenient for a process to have `WITH.MONITOR` at its top level and then do all its interesting waiting using `MONITOR.AWAIT.EVENT`. Not only is this often cleaner, but in the present implementation in cases where the lock is frequently accessed, it saves the `RESETLST` overhead of `WITH.MONITOR`.

Programming restriction: There must not be an `ERRORSET` between the enclosing `WITH.MONITOR` and the call to `MONITOR.AWAIT.EVENT` such that the `ERRORSET` would catch an `ERROR!` and continue inside the monitor, for the lock would not have been reobtained. (The reason for this restriction is that, although `MONITOR.AWAIT.EVENT` won't itself error, you could have caused an error with an interrupt, or a `PROCESS.EVAL` in the context of the waiting process that produced an error.)

On rare occasions it may be useful to manipulate monitor locks directly. The following two functions are used in the implementation of `WITH.MONITOR`:

(OBTAIN.MONITORLOCK *LOCK DONTWAIT UNWINDSAVE*) [Function]

>Takes possession of *LOCK*, waiting if necessary until it is free, unless *DONTWAIT* is true, in which case it returns NIL immediately. If *UNWINDSAVE* is true, performs a RESETSAVE to be unwound when the enclosing RESETLST exits. Returns *LOCK* if *LOCK* was successfully obtained, T if the current process already owned *LOCK*.

(RELEASE.MONITORLOCK *LOCK EVENIFNOTMINE*) [Function]

>Releases *LOCK* if it is owned by the current process, and wakes up the next process, if any, waiting to obtain the lock.

>If *EVENIFNOTMINE* is non-NIL, the lock is released even if it is not owned by the current process.

When a process is deleted, any locks it owns are released.

Global Resources

The biggest source of problems in the multi-processing environment is the matter of global resources. Two processes cannot both use the same global resource if there can be a process switch in the middle of their use (currently this means calls to BLOCK, but ultimately with a preemptive scheduler means anytime). Thus, user code should be wary of its own use of global variables, if it ever makes sense for the code to be run in more than one process at a time. "State" variables private to a process should generally be bound in that process; structures that are shared among processes (or resources used privately but expensive to duplicate per process) should be protected with monitor locks or some other form of synchronization.

Aside from user code, however, there are many *system* global variables and resources. Most of these arise historically from the single-process Interlisp-10 environment, and will eventually be changed in Interlisp-D to behave appropriately in a multi-processing environment. Some have already been changed, and are described below. Two other resources not generally thought of as global variables—the keyboard and the mouse— are particularly idosyncratic, and are discussed in the next section.

The following resources, which are global in Interlisp-10, are allocated per process in Interlisp-D: primary input and output (the streams affected by **INPUT** and **OUTPUT**), terminal input and output (the streams designated by the name **T**), the primary read table and primary terminal table, and dribble files. Thus, each process can print to its own primary output, print to the terminal, read from a different primary input, all without interfering with another process's reading and printing.

Each process begins life with its primary and terminal input/output streams set to a dummy stream. If the process attempts input or output using any of those dummy streams, e.g., by calling (READ T), or (PRINT & T), a tty window is automatically created for the process, and that window becomes the primary input/output and terminal input/output for the process. The default tty window is created at or near the region specified in the variable DEFAULTTTYREGION.

A process can, of course, call TTYDISPLAYSTREAM explicitly to give itself a tty window of its own choosing, in which case the automatic mechanism never comes into play. Calling TTYDISPLAYSTREAM when a process has no tty window not only sets the terminal streams, but also sets the primary input and output streams to be that window, assuming they were still set to the dummy streams.

(HASTTYWINDOWP *PROCESS*)   [Function]

> Returns T if the process *PROCESS* has a tty window; NIL otherwise. If *PROCESS* is NIL, it defaults to the current process.

Other system resources that are typically changed by RESETFORM, RESETLST, or RESETVARS are all global entities. In the multiprocessing environment, these constructs are suspect, as there is no provision for "undoing" them when a process switch occurs. For example, in the current release of Interlisp-D, it is not possible to set the print radix to 8 inside only one process, as the print radix is a global entity.

Note that RESETFORM and similar expressions are perfectly valid in the process world, and even quite useful, when they manipulate things strictly within one process. The process world is arranged so that deleting a process also unwinds any RESETxxx expressions that were performed in the process and are still waiting to be unwound, exactly as if a Control-D had reset the process to the top. Additionally, there is an implicit RESETLST at the top of each process, so that RESETSAVE can be used as a way of providing "cleanup" functions for when a process is deleted. For these, the value of RESETSTATE (see Chapter 14) is NIL if the process finished normally, ERROR if it was aborted by an error, RESET if the process was explicitly deleted, and HARDRESET if the process is being restarted after a HARDRESET or a RESTART.PROCESS.

Typein and the TTY Process

There is one global resource, the keyboard, that is particularly problematic to share among processes. Consider, for example, having two processes both performing (READ T). Since the keyboard input routines block while there is no input, both processes would spend most of their time blocking, and it would simply be a matter of chance which process received each character of typein.

To resolve such dilemmas, the system designates a distinguished process, termed the *tty process*, that is assumed to be the process that is involved in terminal interaction. Any typein from the keyboard goes to that process. If a process other than the tty process requests keyboard input, it blocks until it becomes the tty process. When the tty process is switched (in any of the ways described further below), any typeahead that occurred before the switch is saved and associated with the current tty process. Thus, it is always the case that keystrokes are sent to the process that is the tty process at the time of the keystrokes, regardless of when that process actually gets around to reading them.

It is less immediately obvious how to handle keyboard interrupt characters, as their action is asynchronous and not always tied to typein. Interrupt handling is described in the Handling of Interrupts section below.

Switching the TTY Process

Any process can make itself be the tty process by calling `TTY.PROCESS`.

(`TTY.PROCESS` *PROC*)   [Function]

> Returns the handle of the current tty process. In addition, if *PROC* is non-`NIL`, makes it be the tty process. The special case of *PROC* = `T` is interpreted to mean the executive process; this is sometimes useful when a process wants to explicitly give up being the tty process.

(`TTY.PROCESSP` *PROC*)   [Function]

> True if *PROC* is the tty process; *PROC* defaults to the running process. Thus, (`TTY.PROCESSP`) is true if the caller is the tty process.

(`WAIT.FOR.TTY` *MSECS NEEDWINDOW*)   [Function]

> Efficiently waits until (`TTY.PROCESSP`) is true. `WAIT.FOR.TTY` is called internally by the system functions that read from the terminal; user code thus need only call it in special cases.

> If *MSECS* is non-`NIL`, it is the number of milliseconds to wait before timing out. If `WAIT.FOR.TTY` times out before (`TTY.PROCESSP`) is true, it returns `NIL`, otherwise it returns `T`. If *MSECS* is `NIL`, `WAIT.FOR.TTY` will not time out.

> If *NEEDWINDOW* is non-`NIL`, `WAIT.FOR.TTY` opens a TTY window for the current process if one isn't already open.

> `WAIT.FOR.TTY` spawns a new mouse process if called under the mouse process (see `SPAWN.MOUSE`, in the Keeping the Mouse Alive section below).

In some cases, such as in functions invoked as a result of mouse action or a user's typed-in call, it is reasonable for the function to invoke TTY.PROCESS itself so that it can take subsequent user type in. In other cases, however, this is too undisciplined; it is desirable to let the user designate which process typein should be directed to. This is most conveniently done by mouse action.

The system supports the model that "to type to a process, you click in its window." To cooperate with this model, any process desiring keyboard input should put its process handle as the PROCESS property of its window(s). To handle the common case, the function TTYDISPLAYSTREAM does this automatically when the ttydisplaystream is switched to a new window. A process can own any number of windows; clicking in any of those windows gives the process the tty.

This mechanism suffices for most casual process writers. For example, if a process wants all its input/output interaction to occur in a particular window that it has created, it should just make that window be its tty window by calling TTYDISPLAYSTREAM. Thereafter, it can PRINT or READ to/from the T stream; if the process is not the tty process at the time that it calls READ, it will block until the user clicks in the window.

For those needing tighter control over the tty, the default behavior can be overridden or supplemented. The remainder of this section describes the mechanisms involved.

There is a window property WINDOWENTRYFN that controls whether and how to switch the tty to the process owning a window. The mouse handler, before invoking any normal BUTTONEVENTFN, specifically notices the case of a button going down in a window that belongs to a process (i.e., has a PROCESS window property) that is not the tty process. In this case, it invokes the window's WINDOWENTRYFN of one argument (*WINDOW*). WINDOWENTRYFN defaults to GIVE.TTY.PROCESS:

(GIVE.TTY.PROCESS *WINDOW*)   [Function]

> If *WINDOW* has a PROCESS property, performs (TTY.PROCESS (WINDOWPROP *WINDOW* 'PROCESS)) and then invokes *WINDOW*'s BUTTONEVENTFN function (or RIGHTBUTTONFN if the right button is down).

There are some cases where clicking in a window does not always imply that the user wants to talk to that window. For example, clicking in a text editor window with a shift key held down means to "shift-select" some piece of text into the input buffer of the *current* tty process. The editor supports this by supplying a WINDOWENTRYFN that performs GIVE.TTY.PROCESS if no shift key is down, but goes into its shift-select mode, without changing the tty process, if a shift key is down. The shift-select mode performs a BKSYSBUF of the selected text when the shift key is let up, the BKSYSBUF feeding input to the current tty process.

Sometimes a process wants to be notified when it becomes the tty process, or stops being the tty process. To support this, there are two process properties, TTYEXITFN and TTYENTRYFN. The actions taken by TTY.PROCESS when it switches the tty to a new process are as follows: the former tty process's TTYEXITFN is called with two arguments (*OLDTTYPROCESS NEWTTYPROCESS*); the new process is made the tty process; finally, the new tty process's TTYENTRYFN is called with two arguments (*NEWTTYPROCESS OLDTTYPROCESS*). Normally the TTYENTRYFN and TTYEXITFN need only their first argument, but the other process involved in the switch is supplied for completeness. In the present system, most processes want to interpret the keyboard in the same way, so it is considered the responsibility of any process that changes the keyboard interpretation to restore it to the normal state by its TTYEXITFN.

A window is "owned" by the last process that anyone gave as the window's PROCESS property. Ordinarily there is no conflict here, as processes tend to own disjoint sets of windows (though, of course, cooperating processes can certainly try to confuse each other). The only likely problem arises with that most global of windows, PROMPTWINDOW. Programs should not be tempted to read from PROMPTWINDOW. This is not usually necessary anyway, as the first attempt to read from T in a process that has not set its TTYDISPLAYSTREAM to its own window causes a tty window to be created for the process (see the Global Resources section above).

Handling of Interrupts

At the time that a keyboard interrupt character (see Chapter 30) is struck, any process could be running, and some decision must be made as to which process to actually interrupt. To the extent that keyboard interrupts are related to typein, most interrupts are taken in the tty process; however, the following are handled specially:

RESET (initially Control-D)
ERROR (initially Control-E)   These interrupts are taken in the mouse process, if the mouse is not in its idle state; otherwise they are taken in the tty process. Thus, Control-E can be used to abort some mouse-invoked window action, such as the Shape command. As a consequence, note that if the mouse invokes some lengthy computation that the user thinks of as "background", Control-E still aborts it, even though that may not have been what the user intended. Such lengthy computations, for various reasons, should generally be performed by spawning a separate process to perform them. The RESET interrupt in a process other than the executive is interpreted exactly as if an error unwound the process to its top level: if the process was

	designated RESTARTABLE = T, it is restarted; otherwise it is killed.
HELP (initially Control-G)	A menu of processes is presented to the user, who is asked to select which one the interrupt should occur in. The current tty process appears with a * next to its name at the top of the menu. The menu also includes an entry "[Spawn Mouse]", for the common case of needing a mouse because the mouse process is currently tied up running someone's BUTTONEVENTFN; selecting this entry spawns a new mouse process, and no break occurs.
BREAK (initially Control-B)	Performs the HELP interrupt in the mouse process, if the mouse is not in its idle state; otherwise it is performed in the tty process.
RUBOUT (initially DELETE)	This interrupt clears typeahead in *all* processes.
RAID, STACK OVERFLOW STORAGE FULL	These interrupts always occur in whatever process was running at the time the interrupt struck. In the cases of STACK OVERFLOW and STORAGE FULL, this means that the interrupt is more likely to strike in the offending process (especially if it is a "runaway" process that is not blocking). Note, however, that this process is still not necessarily the guilty party; it could be an innocent bystander that just happened to use up the last of a resource prodigiously consumed by some other process.

Keeping the Mouse Alive

Since the window mouse handler runs in its own process, it is not available while a window's BUTTONEVENTFN function (or any of the other window functions invoked by mouse action) is running. This leads to two sorts of problems: (1) a long computation underneath a BUTTONEVENTFN deprives the user of the mouse for other purposes, and (2) code that runs as a BUTTONEVENTFN cannot rely on other BUTTONEVENTFNs running, which means that there some pieces of code that run differently from normal when run under the mouse process. These problems are addressed by the following functions:

(SPAWN.MOUSE —) [Function]

> Spawns another mouse process, allowing the mouse to run even if it is currently "tied up" under the current mouse process. This function is intended mainly to be typed in at the Lisp executive when the user notices the mouse is busy.

(ALLOW.BUTTON.EVENTS) [Function]

> Performs a (SPAWN.MOUSE) only when called underneath the mouse process. This should be called (once, on entry) by any function that relies on BUTTONEVENTFNs for completion, if there is any possibility that the function will itself be invoked by a mouse function.

It never hurts, at least logically, to call SPAWN.MOUSE or ALLOW.BUTTON.EVENTS needlessly, as the mouse process arranges to quietly kill itself if it returns from the user's BUTTONEVENTFN and finds that another mouse process has sprung up in the meantime. (There is, of course, some computational expense.)

Process Status Window

The background menu command PSW (see Chapter 28) and the function PROCESS.STATUS.WINDOW (below) create a "Process Status Window", that allows the user to examine and manipulate all of the existing processes:

The window consists of two menus. The top menu lists all the processes at the moment. Commands in the bottom menu operate on the process selected in the top menu (EXEC in the example above). The commands are:

BT, BTV, BTV*, BTV!	Displays a backtrace of the selected process.
WHO?	Changes the selection to the tty process, i.e., the one currently in control of the keyboard.
KBD←	Associates the keyboard with the selected process; i.e., makes the selected process be the tty process.
INFO	If the selected process has an INFOHOOK property, calls it. The hook may be a function, which is then applied to two arguments, the process and the button (LEFT or MIDDLE) used to invoke INFO, or a form, which is simply EVAL'ed. The APPLY or EVAL happens in the context of the selected process, using PROCESS.APPLY or PROCESS.EVAL. The INFOHOOK process property can be set using PROCESSPROP (see the Creating and Destroying Processes section above).
BREAK	Enter a break under the selected process. This has the side effect of waking the process with the value returned from the break.
KILL	Deletes the selected process.
RESTART	Restarts the selected process.
WAKE	Wakes the selected process. Prompts for a value to wake it with (see WAKE.PROCESS).
SUSPEND	Suspends the selected process; i.e., causes it to block indefinitely (until explicitly woken).

(PROCESS.STATUS.WINDOW *WHERE*)   [Function]

Puts up a process status window that provides several debugging commands for manipulating running processes. If the window is already up, PROCESS.STATUS.WINDOW refreshes it. If *WHERE* is a position, the window is placed in that position; otherwise, the user is prompted for a position.

Currently, the process status window runs under the mouse process, like other menus, so if the mouse is unavailable (e.g., a mouse function is performing an extensive computation), you may be unable to use the process status window (you can try `SPAWN.MOUSE`, of course).

Non-Process Compatibility

This section describes some considerations for authors of programs that ran in the old single-process Interlisp-D environment, and now want to make sure they run properly in the Multi-processing world. The biggest problem to watch out for is code that runs underneath the mouse handler. Writers of mouse handler functions should remember that in the process world the mouse handler runs in its own process, and hence (a) you cannot depend on finding information on the stack (stash it in the window instead), and (b) while your function is running, the mouse is not available (if you have any non-trivial computation to do, spawn a process to do it, notify one of your existing processes to do it, or use `PROCESS.EVAL` to run it under some other process).

The following functions are meaningful even if the process world is not on: `BLOCK` (invokes the system background routine, which includes handling the mouse); `TTY.PROCESS`, `THIS.PROCESS` (both return `NIL`); and `TTY.PROCESSP` (returns `T`, i.e., anyone is allowed to take tty input).   In addition, the following two functions exist in both worlds:

(`EVAL.AS.PROCESS` *FORM*)   [Function]

> Same as (`ADD.PROCESS` *FORM* `'RESTARTABLE` `'NO`), when processes are running, `EVAL` when not. This is highly recommended for mouse functions that perform any non-trivial activity.

(`EVAL.IN.TTY.PROCESS` *FORM WAITFORRESULT*)   [Function]

> Same as (`PROCESS.EVAL` (`TTY.PROCESS`) *FORM WAITFORRESULT*), when processes are running, `EVAL` when not.

Most of the process functions that do not take a process argument can be called even if processes aren't running. `ADD.PROCESS` creates, but does not run, a new process (it runs when `PROCESSWORLD` is called).

23. STREAMS AND FILES

A stream is an object that provides an interface to a physical or logical device. The stream object contains local data and methods that operate on the stream object.   Medley's general-purpose I/O functions take a stream as one of their arguments. Not every device is capable of implementing every I/O operation, while some devices offer special functions for that device alone. Such restrictions and extensions are noted in the documentation of each device. The majority of the streams used in Medley fall into two categories: file streams and image streams.

A file is a sequence of data stored on some device that allows the data to be retrieved at a later time. Files are identified by a name specifying their storage devices. Input or output to a file is performed through a stream to the file, using OPENSTREAM (below). In addition, there are functions that manipulate the files themselves, rather than their data content.

An image stream is an output stream to a display device, such as the display screen or a printer. In addition to the standard output operations, an image stream implements a variety of graphics operations, such as drawing lines and displaying characters in multiple fonts. Unlike a file, the "content" of an image stream cannot be retrieved. Image streams are described in Chapter 26.

This chapter describes operations specific to file devices: how to name files, how to open streams to files, and how to manipulate files on their devices.

Opening and Closing File Streams

To perform input from or output to a file, you must create a stream to the file, using OPENSTREAM:

(**OPENSTREAM** *FILE ACCESS RECOG PARAMETERS* —)   [Function]

> Opens and returns a stream for the file specified by *FILE*, a file name. *FILE* can be either a string or a symbol. The syntax and manipulation of file names is described at length in the FILENAMES section below. Incomplete file names are interpreted with respect to the connected directory (below).
>
> *RECOG* specifies the recognition mode of *FILE* (below). If *RECOG* = NIL, it defaults according to the value of *ACCESS*.
>
> *ACCESS* specifies the "access rights" to be used when opening the file. Possible values are:

> INPUT
> > Only input operations are permitted on the already existing file. Starts reading at the beginning of the file. *RECOG* defaults to OLD.

> OUTPUT
> > Only output operations are permitted on the initially empty file. Starts writing at the beginning of the file. While the file is open, other users or processes are unable to open the file for either input or output. *RECOG* defaults to NEW.

> BOTH
> > Both input and output operations are permitted on the file.   Starts reading or writing at the beginning of the file. *RECOG* defaults to OLD/NEW. *ACCESS* = BOTH implies random access (Chapter 25), and may not be possible for files on some devices.

APPEND   Only sequential output operations are permitted on the file. Starts writing at the end of the file. *RECOG* defaults to OLD/NEW. *ACCESS* = APPEND may not be allowed for files on some devices.

Note: *ACCESS* = OUTPUT implies that you intend to write a new or different file, even if a version number was specified and the corresponding file already exists. Any previous contents of the file are discarded, and the file is empty immediately after the OPENSTREAM. If you want to write on an already existing file while preserving the old contents, the file must be opened for access BOTH or APPEND.

PARAMETERS is a list of pairs (*ATTRIB VALUE*), where *ATTRIB* is a file attribute (see SETFILEINFO below). A non-list *ATTRIB* in *PARAMETERS* is treated as the pair (*ATTRIB* T). Generally speaking, attributes that belong to the permanent file (e.g., TYPE) can only be set when creating a new file, while attributes that belong only to a particular opening of a file (e.g., ENDOFSTREAMOP) can be set on any call to OPENSTREAM. Not all devices honor all attributes; those not recognized by a particular device are simply ignored.

In addition to the attributes permitted by SETFILEINFO, the following attributes are accepted by OPENSTREAM as values of *ATTRIB* in its *PARAMETERS* argument:

DON'T.CHANGE.DATE   If *VALUE* is non-NIL, the file's creation date is not changed when the file is opened. This option is meaningful only for old files opened for BOTH access. You should use this only for specialized applications where the caller does not want the file system to believe the file's content has been changed.

SEQUENTIAL   If *VALUE* is non-NIL, this opening of the file need support only sequential access; i.e., the caller intends never to use SETFILEPTR. For some devices, sequential access to files is much more efficient than random access. Note that the device may choose to ignore this attribute and still open the file in a manner that permits random access. Also note that this attribute does not make sense with *ACCESS* = BOTH.

If *FILE* is not recognized by the file system, OPENSTREAM causes the error FILE NOT FOUND. Ordinarily, this error is intercepted via an entry on ERRORTYPELST (Chapter 24), which causes SPELLFILE (see the Searching File Directories below) to be called. SPELLFILE searches alternate directories and possibly attempts spelling correction on the file name. Only if SPELLFILE is unsuccessful will the FILE NOT FOUND error actually occur.

If *FILE* exists but cannot be opened, OPENSTREAM causes one of several other errors: FILE WON'T OPEN if the file is already opened for conflicting access by someone else; PROTECTION VIOLATION if the file is protected against the operation; FILE SYSTEM RESOURCES EXCEEDED if there is no more room in the file system.

(**CLOSEF** *FILE*)   [Function]

> Closes *FILE* and returns its full file name. Generates an error, FILE NOT OPEN, if *FILE* does not designate an open stream. After closing a stream, no further input/output operations are permitted on it.

> If *FILE* is NIL, it is defaulted to the primary input stream if that is not the terminal stream, or else the primary output stream if that is not the terminal stream. If both primary input and output streams are the terminal input/output streams, CLOSEF returns NIL. If CLOSEF closes either the primary input stream or the primary output stream (either explicitly or in the *FILE* = NIL case), it resets the primary stream for that direction to be the corresponding terminal stream. See Chapter 25 for information on the primary input/output streams.

> WHENCLOSE (below) allows you to "advise" CLOSEF to perform various operations when a file is closed.

> Because of buffering, the contents of a file open for output are not guaranteed to be written to the actual physical file device until CLOSEF is called. Buffered data can be forced out to a file without closing the file by using the function FORCEOUTPUT (Chapter 25).

> Some network file devices perform their transactions in the background. As a result, it is possible for a file to be closed by CLOSEF and yet not be "fully" closed for a small time period afterward. During this time the file appears to be busy and cannot be opened for conflicting access by others.

(**CLOSEF?** *FILE*)   [Function]

> Closes *FILE* if it is open, returning the value of CLOSEF; otherwise does nothing and returns NIL.

In the present implementation of Medley, all open streams to files are kept in a registry of "open files". This registry does not include nameless streams, such as string streams (below), display streams (Chapter 28), and the terminal input and output streams; nor streams explicitly hidden from you, such as dribble streams (Chapter 30). This registry may not persist in future implementations of Medley, but at the present time it is accessible by the following two functions:

(**OPENP** *FILE ACCESS*)   [Function]

> *ACCESS* is an access mode for a stream opening (see OPENSTREAM), or NIL for any access.

> If *FILE* is a stream, returns its full name if it is open for the specified access, otherwise NIL.

> If *FILE* is a file name (a symbol), *FILE* is processed according to the rules of file recognition (below). If a stream open to a file by that name is registered and open for the specified access, then the file's full name is returned. If the file name is not recognized, or no stream is open to the file with the specified access, NIL is returned.

> If *FILE* is NIL, returns a list of the full names of all registered streams that are open for the specified access.

(**CLOSEALL** *ALLFLG*)   [Function]

> Closes all streams in the value of (OPENP). Returns a list of the files closed.

> WHENCLOSE (below) allows certain files to be "protected" from CLOSEALL. If *ALLFLG* is T, all files, including those protected by WHENCLOSE, are closed.

File Names

A file name in Medley is a string or symbol whose characters specify a "path" to the actual file: on what host or device the file resides, in which directory, and so forth. Because Medley supports a variety of non-local file devices, parts of the path could be device-dependent. However, it is desirable for programs to be able to manipulate file names in a device-independent manner. To this end, Medley specifies a uniform file name syntax over all devices; the functions that perform the actual file manipulation for a particular device are responsible for any translation to that device's naming conventions.

A file name is composed of a collection of *fields*, some of which have specific meanings. The functions described below refer to each field by a *field name*, a literal atom from among the following: HOST, DEVICE, DIRECTORY, NAME, EXTENSION, and VERSION. The standard syntax for a file name is {HOST}DEVICE:<DIRECTORY>NAME.EXTENSION;VERSION. Some host's file systems do not use all of those fields in their file names.

> HOST
> > Specifies the host whose file system contains the file. In the case of local file devices, the "host" is the name of the device, e.g., DSK or FLOPPY.

> DEVICE
> > Specifies, for those hosts that divide their file system's name space among mutiple physical devices, the device or logical structure on which the file resides. This should not be confused with Medley's abstract "file device", which denotes either a host or a local physical device and is specified by the HOST field.

> DIRECTORY
> > Specifies the "directory" containing the file. A directory usually is a grouping of a possibly large set of loosely related files, e.g., the personal files of a particular user, or the files belonging to some project. The DIRECTORY field usually consists of a principal directory and zero or more subdirectories that together describe a path through a file system's hierarchy. Each subdirectory name is set off from the previous directory or subdirectory by the character ">"; e.g., "LISP>LIBRARY>NEW".

> NAME
> > This field carries no specific meaning, but generally names a set of files thought of as being different renditions of the "same" abstract file.

> EXTENSION
> > This field also carries no specific meaning, but generally distinguishes the form of files having the same name. Most files systems have some "conventional" extensions that denote something about the content of the file. For example, in Medley, the extension DCOM, LCOM or DFASL denotes files containing compiled function definitions.

VERSION   A number used to distinguish the versions or "generations" of the files having a common name and extension. The version number is incremented each time a new file by the same name is created.

Most functions that take as input "a directory" accept either a directory name (the contents of the DIRECTORY field of a file name) or a "full" directory specification—a file name fragment consisting of only the fields HOST, DEVICE, and DIRECTORY. In particular, the "connected directory" (see below) consists, in general, of all three fields.

For convenience in dealing with certain operating systems, Medley also recognizes [] and () as host delimiters (synonymous with {}), and / as a directory delimiter (synonymous with < at the beginning of a directory specification and > to terminate directory or subdirectory specification). For example, a file on a Unix file server UNX with the name /usr/foo/bar/stuff.tedit, whose DIRECTORY field is thus usr/foo/bar, could be specified as {UNX}/usr/foo/bar/stuff.tedit, or (UNX)<usr/foo/bar>stuff.tedit, or several other variations. Note that when using [] or () as host delimiters, they usually must be escaped with the reader's escape character if the file name is expressed as a symbol rather than a string.

Different hosts have different requirements for vaild characters in file names. In Medley, all characters are valid. However, in order to be able to parse a file name into its component fields, it is necessary that those characters that are conventionally used as file name delimiters be quoted when they appear inside of fields where there could be ambiguity. The file name quoting character is " ' " (single quote). Thus, the following characters must be quoted when not used as delimiters: >, :, ;, /, and ' itself. The character . (period) need only be quoted if it is to be considered a part of the EXTENSION field. The characters },], and) need only be quoted in a file name when the host field of the name is introduced by {, [, and (, respectively. The characters {, [, (, and < need only be quoted if they appear as the first character of a file name fragment, where they would otherwise be assumed to introduce the HOST or DIRECTORY fields.

The following functions are the standard way to manipulate file names in Medley. Their operation is purely syntactic—they perform no file system operations themselves.

(**UNPACKFILENAME.STRING** *FILENAME*)   [Function]

Parses *FILENAME*, returning a list in property list format of alternating field names and field contents. The field contents are returned as strings. If it is a stream, its full name is used.

Only those fields actually present in *FILENAME* are returned. A field is considered present if its delimiting punctuation is present, even if the field itself is empty. Empty fields are denoted by "" (the empty string).

Examples:

```
(UNPACKFILENAME.STRING "FOO.BAR") =>
   (NAME "FOO" EXTENSION "BAR")

(UNPACKFILENAME.STRING "FOO.;2") =>
   (NAME "FOO" EXTENSION "" VERSION "2")

(UNPACKFILENAME.STRING "FOO;") =>
   (NAME "FOO" VERSION "")

(UNPACKFILENAME.STRING
```

```
                      "{ERIS}<LISP>CURRENT>IMTRAN.DCOM;21")
                => (HOST "ERIS" DIRECTORY "LISP>CURRENT"
                      NAME "IMTRAN" EXTENSION "DCOM"
                      VERSION "21")
```

(**UNPACKFILENAME** *FILE*)   [Function]

> Old version of UNPACKFILENAME.STRING that returns the field values as atoms, rather
> than as strings. UNPACKFILENAME.STRING is now considered the "correct" way of
> unpacking file names, because it does not lose information when the contents of a field are
> numeric. For example,
>
> ```
> (UNPACKFILENAME 'STUFF.TXT) =>
>     (NAME STUFF EXTENSION TXT)
> ```
>
> but
>
> ```
> (UNPACKFILENAME 'STUFF.029) =>
>     (NAME STUFF EXTENSION 29)
> ```
>
> Explicitly omitted fields are denoted by the atom NIL, rather than the empty string.
>
>> Note: Both UNPACKFILENAME and UNPACKFILENAME.STRING leave the
>> trailing colon on the device field, so that the Tenex device NIL: can be
>> distinguished from the absence of a device. Although
>> UNPACKFILENAME.STRING is capable of making the distinction, it
>> retains this behavior for backward compatibility. Thus,
>>
>> ```
>> (UNPACKFILENAME.STRING '{TOAST}DSK:FOO) =>
>>     (HOST "TOAST" DEVICE "DSK:" NAME "FOO")
>> ```

(**FILENAMEFIELD** *FILENAME FIELDNAME*)   [Function]

> Returns, as an atom, the contents of the *FIELDNAME* field of *FILENAME*. If *FILENAME* is a
> stream, its full name is used.

(**PACKFILENAME.STRING** *FIELD$_1$ CONTENTS$_1$ ... FIELD$_N$ CONTENTS$_N$*)   [NoSpread
Function]

> Takes a sequence of alternating field names and field contents (atoms or strings), and
> returns the corresponding file name, as a string.
>
> If PACKFILENAME.STRING is given a single argument, it is interpreted as a list of
> alternating field names and field contents. Thus PACKFILENAME.STRING and
> UNPACKFILENAME.STRING operate as inverses.
>
> If the same field name is given twice, the *first* occurrence is used.
>
> The contents of the field name DIRECTORY may be either a directory name or a full
> directory specification as described above.
>
> PACKFILENAME.STRING also accepts the "field name" BODY to mean that its contents
> should itself be unpacked and spliced into the argument list at that point. This feature, in
> conjunction with the rule that fields early in the argument list override later duplicates, is
> useful for altering existing file names. For example, to provide a default field, place BODY

first in the argument list, then the default fields. To override a field, place the new fields
first and BODY last.

If the value of the BODY field is a stream, its full name is used.

Examples:

```
(PACKFILENAME.STRING 'DIRECTORY "LISP"
    'NAME "NET")
       =>  "<LISP>NET"
(PACKFILENAME.STRING 'NAME "NET"
    'DIRECTORY "{DSK}<LISPFILES>")
       => "{DSK}<LISPFILES>NET"
(PACKFILENAME.STRING 'DIRECTORY "{DSK}"
    'BODY "{TOAST}<FOO>BAR")
       => "{DSK}BAR"
(PACKFILENAME.STRING 'DIRECTORY "FRED"
    'BODY "{TOAST}<FOO>BAR")
       => "{TOAST}<FRED>BAR"
(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR"
    'DIRECTORY "FRED")
       => "{TOAST}<FOO>BAR"
(PACKFILENAME.STRING 'VERSION NIL
    'BODY "{TOAST}<FOO>BAR.DCOM;2")
       => "{TOAST}<FOO>BAR.DCOM"
(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR.DCOM"
    'VERSION 1)
       => "{TOAST}<FOO>BAR.DCOM;1"
(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR.DCOM;"
    'VERSION 1)
       => "{TOAST}<FOO>BAR.DCOM;"
(PACKFILENAME.STRING 'BODY "BAR.;1"
    'EXTENSION "DCOM")
       => "BAR.;1"
(PACKFILENAME.STRING 'BODY "BAR;1"
    'EXTENSION "DCOM")
       => "BAR.DCOM;1"
```

In the last two examples, note that in one case the extension is explicitly present in the
body (as indicated by the preceding period), while in the other there is no indication of an
extension, so the default is used.

($\textbf{PACKFILENAME}$ $FIELD_1$ $CONTENTS_1$ ... $FIELD_N$ $CONTENTS_N$)   [NoSpread Function]

The same as PACKFILENAME.STRING, except that it returns the file name as a symbol,
instead of a string.

Incomplete File Names

In general, it is not necessary to pass a complete file name (one containing all the fields listed above) to
functions that take a file name as an argument. Interlisp supplies suitable defaults for certain fields
(below). Functions that return names of actual files, however, always return the full file name.

If the version field is omitted from a file name, Interlisp performs version recognition, as described below.

If the host, device and/or directory field are omitted from a file name, Interlisp uses the currently connected directory. You can change the currently connected directory by by calling `CNDIR` (below) or using the programmer's assistant command `CONN`.

Defaults are added to the partially specified name "left to right" until a host, device or directory field is encountered. Thus, if the connected directory is {TWENTY}PS:<FRED>, then

>   BAR.DCOM means
>   {TWENTY}PS:<FRED>BAR.DCOM
>
>   <GRANOLA>BAR.DCOM means
>   {TWENTY}PS:<GRANOLA>BAR.DCOM
>
>   MTA0:<GRANOLA>BAR.DCOM means
>   {TWENTY}MTA0:<GRANOLA>BAR.DCOM
>
>   {THIRTY}<GRANOLA>BAR.DCOM means
>   {THIRTY}<GRANOLA>BAR.DCOM

In addition, if the partially specified name contains a subdirectory, but no principal directory, then the subdirectory is appended to the connected directory. For example,

>   ISO>BAR.DCOM means
>   {TWENTY}PS:<FRED>ISO>BAR.DCOM

Or, if the connected directory is the Unix directory {UNX}/usr/fred/, then iso/bar.dcom means {UNX}/usr/fred/iso/bar.dcom, but /other/bar.dcom means {UNX}/other/bar.dcom.

(**CNDIR** *HOST/DIR*)   [Function]

> Connects to the directory *HOST/DIR*, which can either be a directory name or a full directory specification including host and/or device. If the specification includes just a host, and the host supports directories, the directory is defaulted to the value of (USERNAME); if the host is omitted, connection is made to another directory on the same host as before. If *HOST/DIR* is NIL, connects to the value of LOGINHOST/DIR.

> CNDIR returns the full name of the now-connected directory. Causes an error, Non-existent directory, if *HOST/DIR* is not a valid directory.

> Note that CNDIR does not necessarily require or provide any directory access privileges. Access privileges are checked when a file is opened.

CONN *HOST/DIR*   [Prog. Asst. Command]

> Command form of CNDIR for use at the executive. Connects to *HOST/DIR*, or to the value of LOGINHOST/DIR if *HOST/DIR* is omitted. This command is undoable. —Undoing it causes the system to connect to the previously connected directory.

LOGINHOST/DIR   [Variable]

> CONN with no argument connects to the value of the variable LOGINHOST/DIR, initially {DSK}, but usually reset in your greeting file (Chapter 12).

(**DIRECTORYNAME** *DIRNAME STRPTR*)   [Function]

> If *DIRNAME* is T, returns the full specification of the currently connected directory. If *DIRNAME* is NIL, returns the value of LOGINHOST/DIR. For any other value of *DIRNAME*, returns a full directory specification if *DIRNAME* designates an existing directory (satisfies DIRECTORYNAMEP), otherwise NIL.

> If *STRPTR* is T, the value is returned as an atom, otherwise it is returned as a string.

(**DIRECTORYNAMEP** *DIRNAME HOSTNAME*)   [Function]

> Returns T if *DIRNAME* is a valid directory on host *HOSTNAME*, or on the host of the currently connected directory if *HOSTNAME* is NIL. *DIRNAME* may be either a directory name or a full directory specification containing host and/or device.

> If *DIRNAME* includes subdirectories, this function may or may not pass judgment on their validity. Some hosts support "true" subdirectories, distinct entities manipulable by the file system, while others only provide them as a syntactic convenience.

(**HOSTNAMEP** *NAME*)   [Function]

> Returns T if *NAME* is recognized as a valid host or file device name at the moment HOSTNAMEP is called.

Version Recognition

Most of the file devices in Interlisp support file version numbers. That is, you can have several files of the exact same name, differing only in their VERSION field, which is incremented for each new "version" of the file that is created. When the filesystem encounters a file name without a version number, it must figure out which version was intended. This process is known as *version recognition*.

When OPENSTREAM opens a file for input and no version number is given, the highest existing version number is used. Similarly, when a file is opened for output and no version number is given, a new file is created with a version number one higher than the highest one currently in use with that file name. You can change he version number defaulting for OPENSTREAM by specifying a different value for its *RECOG* argument (see FULLNAME below).

Other functions that accept file names as arguments generally perform default version recognition, which is newest version for existing files, or a new version if using the file name to create a new file. The one exception is DELFILE, which uses the oldest existing version of the file.

The functions below can be used to perform version recognition without actually calling OPENSTREAM to open the file. Note that these functions only tell the truth at the moment they are called, and thus cannot be used to anticipate the name of the file opened by a comparable OPENSTREAM. They are best used as helpful hints.

(**FULLNAME** *X RECOG*)   [Function]

> If *X* is an open stream, simply returns the full file name of the stream. Otherwise, if *X* is a file name given as a string or symbol, performs version recognition, as follows:

If *X* is recognized in the recognition mode specified by *RECOG* as an abbreviation for some file, returns the file's full name, otherwise NIL. *RECOG* is one of the following:

OLD   Chooses the newest existing version of the file. Returns NIL if no file named *X* exists.

OLDEST   Chooses the oldest existing version of the file. Returns NIL if no file named *X* exists.

NEW   Chooses a new version of the file. If versions of *X* already exist, then chooses a version number one higher than highest existing version; otherwise chooses version 1. For some file systems, FULLNAME returns NIL if you do not have the access rights necessary to create a new file named *X*.

OLD/NEW   Tries OLD, then NEW. Choose the newest existing version of the file, if any; otherwise chooses version 1. This usually only makes sense if you intend to open *X* for access BOTH.

RECOG = NIL defaults to OLD. For all other values of *RECOG*, generates an error ILLEGAL ARG.

If *X* already contains a version number, the *RECOG* argument will never change it. In particular, *RECOG* = NEW does not require that the file actually be new. For example, (FULLNAME 'FOO.;2 'NEW) may return {ERIS}<LISP>FOO.;2 if that file already exists, even though (FULLNAME 'FOO 'NEW) would default the version to a new number, perhaps returning {ERIS}<LISP>FOO.;5.

(**INFILEP** *FILE*)   [Function]

Equivalent to (FULLNAME *FILE* 'OLD). Returns the full file name of the newest version of *FILE* if *FILE* is the name of an existing file that can be opened for input, NIL otherwise.

(**OUTFILEP** *FILE*)   [Function]

Equivalent to (FULLNAME *FILE* 'NEW).

Note that INFILEP, OUTFILEP and FULLNAME do not open any files; they are pure predicates. They are also only hints, as they do not imply that the caller has access rights to the file. For example, INFILEP might return non-NIL, but OPENSTREAM might fail for the same file because you don't have read access to it, or the file is open for output by another user. Similarly, OUTFILEP could return non-NIL, but OPENSTREAM could fail with a FILE SYSTEM RESOURCES EXCEEDED error.

Note also that in a shared file system, such as a remote file server, intervening file operations by another user could contradict the information returned by recognition. For example, a file that was INFILEP might be deleted, or between an OUTFILEP and the subsequent OPENSTREAM, another user might create a new version or delete the highest version, causing OPENSTREAM to open a different version of the file than the one returned by OUTFILEP. In addition, some file servers do not support recognition of files in output context. Thus, the "truth" about a file can only be obtained by actually opening the file; creators of files should rely on the name of the stream opened by OPENSTREAM, not

the value returned from these recognition functions. In particular, programmers are discouraged from using `OUTFILEP` or `(FULLNAME NAME 'NEW)`.

Using File Names Instead of Streams

In earlier implementations of Interlisp, from the days of Interlisp-10 onward, the "handle" used to refer to an open file was not a stream, but rather the file's full name, represented as a symbol. When the file name was passed to any I/O function, it was mapped to a stream by looking it up in a list of open files. This scheme was sometimes convenient for typing in file commands at the executive, but was poor for serious programming in two ways. First, mapping from file name to stream on every input/output operation is inefficient. Second, and more importantly, using the file name as the handle on an open stream means that it is not possible to have more than one stream open on a given file at once.

As of this writing, Medley is in a transition period, where it still supports the use of symbol file names as synonymous with open streams, but this use is not recommended. The remainder of this section discusses this usage of file names for the benefit of those reading older programs and wishing to convert them to work properly when this compatibility feature is removed.

File Name Efficiency Considerations

It is possible for a program to be seriously inefficient using a file name as a stream if the program is not using the name returned by `OPENFILE` (below). Any time that an input/output function is called with a file name other than the full file name, Interlisp must perform recognition on the partial file name to determine which open file is intended. Thus if repeated operations are to be performed, it is considerably more efficient to use the full file name returned from `OPENFILE`.

There is a more subtle problem with partial file names, in that recognition is performed on your entire directory, not just the open files. It is possible for a file name that previously denoted one file to suddenly denote a different file. For example, suppose a program performs `(INFILE 'FOO)`, opening `FOO.;1`, and reads several expressions from `FOO`. Then you interrupt the program, create a `FOO.;2` and resume the program (or a user at another workstation creates a `FOO.;2`). Now a call to `READ` giving it `FOO` as its `FILE` argument will generate a `FILE NOT OPEN` error, because `FOO` will be recognized as `FOO.;2`.

Obsolete File Opening Functions

The following functions are now obsolete, but are provided for backwards compatibility:

(**OPENFILE** *FILE ACCESS RECOG PARAMETERS*)   [Function]

> Opens *FILE* with access rights as specified by *ACCESS*, and recognition mode *RECOG*, and returns the full name of the resulting stream. Equivalent to `(FULLNAME (OPENSTREAM FILE ACCESS RECOG PARAMETERS))`.

(**INFILE** *FILE*)   [Function]

> Opens *FILE* for input, and sets it as the primary input stream. Equivalent to `(INPUT (OPENSTREAM FILE 'INPUT 'OLD))`

(**OUTFILE** *FILE*) [Function]

> Opens *FILE* for output, and sets it as the primary output stream. Equivalent to `(OUTPUT (OPENSTREAM FILE 'OUTPUT 'NEW))`.

(**IOFILE** *FILE*) [Function]

> Opens *FILE* for both input and output. Equivalent to `(OPENFILE FILE 'BOTH 'OLD)`. Does not affect the primary input or output stream.

Converting Old Programs

At some point in the future, the Medley file system will change so that each call to `OPENSTREAM` returns a distinct stream, even if a stream is already open to the specified file. This change is required in order to deal with files in a multiprocessing environment.

This change will produce the following incompatibilities:

1. The functions `OPENFILE`, `INPUT`, and `OUTPUT` will return a stream, not a full file name. To make this less confusing in interactive situations, streams will have a print format that reveals the underlying file's actual name.

2. Passing anything other than the object returned from `OPENFILE` to I/O operations will cause problems. Passing the file's name will be significantly slower than passing the stream (even when passing the "full" file name), and in the case where there is more than one stream open on the file it might even act on the wrong one.

3. `OPENP` will return `NIL` when passed the name of a file rather than the value of `OPENFILE` or `OPENSTREAM`.

You should consider the following advice when writing new programs and editing existing programs, so your programs will behave properly when the change occurs:

Because of the efficiency and ambiguity considerations described earlier, users have long been encouraged to use only full file names as *FILE* arguments to I/O operations. The "proper" way to have done this was to bind a variable to the value returned from `OPENFILE` and pass that variable to all I/O operations; such code will continue to work. A less proper way to obtain the full file name, but one which has to date not incurred any obvious penalty, is that which binds a variable to the result of an `INFILEP` and passes that to `OPENFILE` and all I/O operations. This has worked because `INFILEP` and `OPENFILE` both return a full file name, an invalid assumption in this future world. Such code should be changed to pass around the value of the `OPENFILE`, not the `INFILEP`.

Code that calls `OPENP` to test whether a possibly incomplete file name is already open should be recoded to pass to `OPENP` only the value returned from `OPENFILE` or `OPENSTREAM`.

Code that uses ordinary string functions to manipulate file names, and in particular the value returned from `OPENFILE`, should be changed to use the the functions `UNPACKFILENAME.STRING` and `PACKFILENAME.STRING`. Those functions work both on file names (strings) and streams (coercing the stream to the name of its file).

Code that tests the value of `OUTPUT` for equality to some known file name or `T` should be examined carefully and, if possible, recoded.

To see more directly the effects of passing around streams instead of file names, replace your calls to OPENFILE with calls to OPENSTREAM. OPENSTREAM is called in exactly the same way, but returns a STREAM. Streams can be passed to READ, PRINT, CLOSEF, etc just as the file's full name can be currently, but using them is more efficient. The function FULLNAME, when applied to a stream, returns its full file name.

Using Files with Processes

Because Medley does not yet support multiple streams per file, problems can arise if different processes attempt to access the same file. You have to be careful not to have two processes manipulating the same file at the same time, since the two processes will be sharing a single input stream and file pointer. For example, you can't have one process TCOMPL a file while another process is running LISTFILES on it.

File Attributes

Any file has a number of "file attributes", such as the read date, protection, and bytesize. The exact attributes that a file can have is dependent on the file device. The functions GETFILEINFO and SETFILEINFO allow you to access file attributes:

(**GETFILEINFO** *FILE ATTRIB*)   [Function]

> Returns the current setting of the *ATTRIB* attribute of *FILE*.

(**SETFILEINFO** *FILE ATTRIB VALUE*)   [Function]

> Sets the attribute *ATTRIB* of *FILE* to be *VALUE*. SETFILEINFO returns T if it is able to change the attribute *ATTRIB*, and NIL if unsuccessful, either because the file device does not recognize *ATTRIB* or because the file device does not permit the attribute to be modified.

The *FILE* argument to GETFILEINFO and SETFILEINFO can be an open stream (or an argument designating an open stream, see Chapter 25), or the name of a closed file. SETFILEINFO in general requires write access to the file.

The attributes recognized by GETFILEINFO and SETFILEINFO fall into two categories: *permanent* attributes, which are properties of the file, and *temporary* attributes, which are properties only of an open stream to the file. The temporary attributes are only recognized when *FILE* designates an open stream; the permanent attributes are usually equally accessible for open and closed files. However, some devices are willing to change the value of certain attributes of an open stream only when specified in the PARAMETERS argument to OPENSTREAM (see above), not on a later call to SETFILEINFO.

The following are permanent attributes of a file:

> BYTESIZE   The byte size of the file. Medley currently only supports byte size 8.
>
> LENGTH   The number of bytes in the file. Alternatively, the byte position of the end-of-file. Like (GETEOFPTR *FILE*), but *FILE* does not have to be open.

SIZE	The size of *FILE* in pages.
CREATIONDATE	The date and time, as a string, that the content of *FILE* was "created". The creation date changes whenever the content of the file is modified, but remains unchanged when a file is transported, unmodified, across file systems. Specifically, COPYFILE and RENAMEFILE (see below) preserve the file's creation date. Note that this is different from the concept of "creation date" used by some operating systems (e.g., Tops20).
WRITEDATE	The date and time, as a string, that the content of *FILE* was last written to this particular file system. When a file is copied, its creation date does not change, but its write date becomes the time at which the copy is made.
READDATE	The date and time, as a string, that *FILE* was last read, or NIL if it has never been read.
ICREATIONDATE IWRITEDATE IREADDATE	The CREATIONDATE, WRITEDATE and READDATE, respectively, in integer form, as IDATE (Chapter 12) would return. This form is useful for comparing dates.
AUTHOR	The name of the user who last wrote the file.
TYPE	The "type" of the file, some indication of the nature of the file's content. The "types" of files allowed depends on the file device. Most devices recognize the symbol TEXT to mean that the file contains just characters, or BINARY to mean that the file contains arbitrary data.

Some devices support a wider range of file types that distinguish among the various sorts of files one might create whose content is "binary". All devices interpret any value of TYPE that they do not support to be BINARY. Thus, GETFILEINFO may return the more general value BINARY instead of the original type that was passed to SETFILEINFO or OPENSTREAM. Similarly, COPYFILE, while attempting to preserve the TYPE of the file it is copying, may turn, say, an INTERPRESS file into a mere BINARY file.

The way in which some file devices (e.g., Xerox file servers) support a wide range of file types is by representing the type as an integer, whose interpretation is known by the client. The variable FILING.TYPES is used to associate symbolic types with numbers for these devices. This list initially contains some of the well-known assignments of type name to number; you can add additional elements to handle any private file types. For example, suppose there existed an NS file type MAZEFILE with numeric value 5678. You could add the element (MAZEFILE 5678) to FILING.TYPES and then use MAZEFILE as a value for the TYPE attribute to SETFILEINFO or OPENSTREAM. Other devices are, of

course, free to store `TYPE` attributes in whatever manner they wish, be it numeric or symbolic. `FILING.TYPES` is merely considered the official registry for Xerox file types.

For most file devices, the `TYPE` of a newly created file, if not specified in the *PARAMETERS* argument to `OPENSTREAM`, defaults to the value of `DEFAULTFILETYPE`, initially `TEXT`.

The following are currently recognized as temporary attributes of an open stream:

`ACCESS`   The current access rights of the stream (see the beginning of this chapter). Can be one of `INPUT`, `OUTPUT`, `BOTH`, `APPEND`; or `NIL` if the stream is not open.

`ENDOFSTREAMOP`   The action to be taken when a stream is at "end of file" and an attempt is made to take input from it. The value of this attribute is a function of one argument, the stream. The function can examine the stream and its calling context and take any action it wishes. If the function returns normally, its should return either `T`, meaning to try the input operation again, or the byte that `BIN` would have returned had there been more bytes to read. Ordinarily, one should not let the `ENDOFSTREAMOP` function return unless one is only performing binary input from the file, since there is no way in general of knowing in what state the reader was at the time the end of file occurred, and hence how it will interpret a single byte returned to it.

The default `ENDOFSTREAMOP` is a system function that causes the error `END OF FILE`. The behavior of that error can be further modified for a particular stream by using the `EOF` option of `WHENCLOSE` (see below).

`EOL`   The end-of-line convention for the stream. This can be `CR`, `LF`, or `CRLF`, indicating with what byte or sequence of bytes the "End Of Line" character is represented on the stream. On input, that sequence of bytes on the stream is read as `(CHARCODE EOL)` by `READCCODE` or the string reader. On output, `(TERPRI)` and `(PRINTCCODE (CHARCODE EOL))` cause that sequence of bytes to be placed on the stream.

The end of line convention is usually not apparent to you. The file system is usually aware of the convention used by a particular remote operating system, and sets this attribute accordingly. If you believe a file actually is stored with a different convention than the default, it is possible to modify the default behavior by including the `EOL` attribute in the *PARAMETERS* argument to `OPENSTREAM`.

`BUFFERS`   Value is the number of 512-byte buffers that the stream maintains at one time. This attribute is only used by certain random-access devices (currently, the local disk, floppy, and Leaf servers); all others ignore it.

Streams open to files generally maintain some portion of the file buffered in memory, so that each call to an I/O function does not

require accessing the actual file on disk or a file server. For files being read or written sequentially, not much buffer space is needed, since once a byte is read or written, it will never need to be seen again. In the case of random access streams, buffering is more complicated, since a program may jump around in the file, using SETFILEPTR (Chapter 25). In this case, the more buffer space the stream has, the more likely it is that after a SETFILEPTR to a place in the file that has already been accessed, the stream still has that part of the file buffered and need not go out to the device again. This benefit must, of course, be traded off against the amount of memory consumed by the buffers.

NS servers implement the following additional attributes for GETFILEINFO (neither of these attributes are settable with SETFILEINFO):

READER   The name of the user who last read the file.

PROTECTION   A list specifying the access rights to the file. Each element of the list is of the form (name nametype . rights). Name is the name of a user or group or a name pattern. Rights is one or more of the symbols ALL READ WRITE DELETE CREATE or MODIFY. For servers running services 10.0 or later, nametype is the symbol "--". , In earlyer releases it is one of the symbols INDIVIDUAL or GROUP

Closing and Reopening Files

The function WHENCLOSE permits you to associate certain operations with open streams that govern how and when the stream will be closed. You can specify that certain functions will be executed before CLOSEF closes the stream and/or after CLOSEF closes the stream. You can make a particular stream be invisible to CLOSEALL, so that it will remain open across user invocations of CLOSEALL.

(**WHENCLOSE** *FILE PROP$_1$ VAL$_1$ . . . PROP$_N$ VAL$_N$*)   [NoSpread Function]

FILE must designate an open stream other than T (NIL defaults to the primary input stream, if other than T, or primary output stream if other than T). The remaining arguments specify properties to be associated with the full name of *FILE*. WHENCLOSE returns the full name of *FILE* as its value.

WHENCLOSE recognizes the following property names:

BEFORE   *VAL* is a function that CLOSEF will apply to the stream just before it is closed. This might be used, for example, to copy information about the file from an in-core data structure to the file just before it is closed.

AFTER   *VAL* is a function that CLOSEF will apply to the stream just after it is closed. This capability permits in-core data structures that know about the stream to be cleaned up when the stream is closed.

CLOSEALL   *VAL* is either YES or NO and determines whether *FILE* will be closed by CLOSEALL (YES) or whether CLOSEALL will ignore it (NO). CLOSEALL

uses `CLOSEF`, so that any `AFTER` functions will be executed if the stream is in fact closed. Files are initialized with `CLOSEALL` set to `YES`.

EOF   *VAL* is a function that will be applied to the stream when an end-of-file error occurs, and the `ERRORTYPELST` entry for that error, if any, returns `NIL`. The function can examine the context of the error, and can decide whether to close the stream, `RETFROM` some function, or perform some other computation. If the function supplied returns normally (i.e., does not `RETFROM` some function), the normal error machinery will be invoked.

The default `EOF` behavior, unless overridden by this `WHENCLOSE` option, is to call the value of `DEFAULTEOFCLOSE` (below).

For some applications, the `ENDOFSTREAMOP` attribute (see above) is a more useful way to intercept the end-of-file error. The `ENDOFSTREAMOP` attribute comes into effect before the error machinery is ever activated.

Multiple `AFTER` and `BEFORE` functions may be associated with a file; they are executed in sequence with the most recently associated function executed first. The `CLOSEALL` and `EOF` values, however, will override earlier values, so only the last value specified will have an effect.

DEFAULTEOFCLOSE   [Variable]

Value is the name of a function that is called by default when an end of file error occurs and no `EOF` option has been specified for the stream by `WHENCLOSE`. The initial value of `DEFAULTEOFCLOSE` is `NILL`, meaning take no special action (go ahead and cause the error). Setting it to `CLOSEF` would cause the stream to be closed before the rest of the error machinery is invoked.

I/O Operations to and from Strings

It is possible to treat a string as if it were the contents of a file by using the following function:

(**OPENSTRINGSTREAM** *STR ACCESS*)   [Function]

Returns a stream that can be used to access the characters of the string *STR*. *ACCESS* may be either `INPUT`, `OUTPUT`, or `BOTH`; `NIL` defaults to `INPUT`. The stream returned may be used exactly like a file opened with the same access, except that output operations may not extend past the end of the original string. Also, string streams do not appear in the value of (`OPENP`).

For example, after performing

```
(SETQ STRM (OPENSTRINGSTREAM "THIS 2 (IS A LIST)"))
```

the following succession of reads could occur:

```
            (READ STRM)  => THIS
            (RATOM STRM) => 2
            (READ STRM)  => (IS A LIST)
            (EOFP STRM)  => T
```

Compatibility Note: In Interlisp-10 it was possible to take input from a string simply by passing the string as the *FILE* argument to an input function. In order to maintain compatibility with this feature, Medley provides the same capability. This not terribly clean feature persists in the present implementation to give users time to convert old code. This means that strings are *not* equivalent to symbols when specifying a file name as a stream argument. In a future release, the old Interlisp-10 string-reading feature will be decommissioned, and OPENSTRINGSTREAM will be the only way to perform I/O on a string.

Temporary Files and the CORE Device

Many operating systems have a notion of "scratch file", a file typically used as temporary storage for data most naturally maintained in the form of a file, rather than some other data structure. A scratch file can be used as a normal file in most respects, but is automatically deleted from the file system after its useful life is up, e.g., when the job terminates, or you log out. In normal operation, you need never explicitly delete such files, since they are guaranteed to disappear soon.

A similar functionality is provided in Medley by core-resident files. Core-resident files are on the device CORE. The directory structure for this device and all files on it are represented completely within your virtual memory. These files are treated as ordinary files by all file operations; their only distinguishing feature is that all trace of them disappears when the Medley image is abandoned.

Core files are opened and closed by name the same as any other file, e.g., (OPENSTREAM '{CORE}<FOO>FIE.DCOM 'OUTPUT). Directory names are completely optional, so files can also have names of the form {CORE}NAME.EXT. Core files can be enumerated by DIRECTORY (see below). While open, they are registered in (OPENP). They do consume virtual memory space, which is only reclaimed when the file is deleted. Some caution should thus be used when creating large CORE files. Since the virtual memory of an Medley workstation usually persists far longer than the typical process on a mainframe computer, it is still important to delete CORE files after they are no longer in use.

For many applications, the name of the scratch file is irrelevant, and there is no need for anyone to have access to the file independent of the program that created it. For such applications, NODIRCORE files are preferable. Files created on the device lisp NODIRCORE are core-resident files that have no name and are registered in no directory. These files "disappear", and the resources they consume are reclaimed, when all pointers to the file are dropped. Hence, such files need never be explicitly deleted or, for that matter, closed. The "name" of such a file is simply the stream object returned from (OPENSTREAM '{NODIRCORE} 'OUTPUT), and it is this stream object that must be passed to all input/output operations, including CLOSEF and any calls to OPENSTREAM to reopen the file.

(**COREDEVICE** *NAME NODIRFLG*)   [Function]

Creates a new device for core-resident files and assigns *NAME* as its device name. Thus, after performing (COREDEVICE 'FOO), one can execute (OPENSTREAM '{FOO}BAR 'OUTPUT) to open a file on that device. Medley is initialized with the single core-resident device named CORE, but COREDEVICE may be used to create any number of logically distinct core devices.

If *NODIRFLG* is non-NIL, a core device that acts like {NODIRCORE} is created.

Compatibility note: In Interlisp-10, it was possible to create scratch files by using file names with suffixes ;S or ;T. In Medley, these suffixes in file names are simply ignored when output is directed to a particular host or device. However, the function PACKFILENAME.STRING is defined to default the device name to CORE if the file has the TEMPORARY attribute and no explicit host is provided.

NULL Device

The NULL device provides a source of content-free "files". (OPENSTREAM '{NULL} 'OUTPUT) creates a stream that discards all output directed at it. (OPENSTREAM '{NULL} 'INPUT) creates a stream that is perpetually at end-of-file (i.e., has no input).

Deleting, Copying, and Renaming Files

(**DELFILE** *FILE*)   [Function]

Deletes *FILE* if possible. The file must be closed. Returns the full name of the file if deleted, else NIL. Recognition mode for *FILE* is OLDEST, i.e., if *FILE* does not have a version number specified, then DELFILE deletes the oldest version of the file.

(**COPYFILE** *FROMFILE* *TOFILE*)   [Function]

Copies *FROMFILE* to a new file named *TOFILE*. The source and destination may be on any combination of hosts/devices. COPYFILE attempts to preserve the TYPE and CREATIONDATE where possible. If the original file's file type is unknown, COPYFILE attempts to infer the type (file type is BINARY if any of its 8-bit bytes have their high bit on).

COPYFILE uses COPYCHARS (Chapter 25) if the source and destination hosts have different EOL conventions. Thus, it is possible for the source and destination files to be of different lengths.

(**RENAMEFILE** *OLDFILE* *NEWFILE*)   [Function]

Renames *OLDFILE* to be *NEWFILE*. Causes an error, FILE NOT FOUND if *FILE* does not exist. Returns the full name of the new file, if successful, else NIL if the rename cannot be performed.

If *OLDFILE* and *NEWFILE* are on the same host/device, and the device implements a renaming primitive, RENAMEFILE can be very fast. However, if the device does not know how to rename files in place, or if *OLDFILE* and *NEWFILE* are on different devices, RENAMEFILE works by copying *OLDFILE* to *NEWFILE* and then deleting *OLDFILE*.

Searching File Directories

DIRECTORIES [Variable]

Global variable containing the list of directories searched (in order) by SPELLFILE and FINDFILE (below) when not given an explicit *DIRLST* argument. In this list, the atom NIL stands for the login directory (the value of LOGINHOST/DIR), and the atom T stands for the currently connected directory. Other elements should be *full* directory specifications, e.g., {TWENTY}PS:<LISPUSERS>, not merely LISPUSERS.

LISPUSERSDIRECTORIES [Variable]

Global variable containing a list of directories to search for "library" package files. Used by the FILES file package command (Chapter 17).

(**SPELLFILE** *FILE NOPRINTFLG NSFLG DIRLST*) [Function]

Searches for the file name *FILE*, possibly performing spelling correction (see Chapter 20). Returns the corrected file name, if any, otherwise NIL.

If *FILE* has a directory field, SPELLFILE attempts spelling correction against the files in that particular directory. Otherwise, SPELLFILE searches for the file on the directory list *DIRLST* before attempting any spelling correction.

If *NOPRINTFLG* is NIL, SPELLFILE asks you to confirm any spelling correction done, and prints out any files found, even if spelling correction is not done. If *NOPRINTFLG* = T, SPELLFILE does not do any printing, nor ask for approval.

If *NSFLG* = T (or NOSPELLFLG = T, see Chapter 20), no spelling correction is attempted, though searching through *DIRLST* still occurs.

DIRLST is the list of directories searched if *FILE* does not have a directory field. If *DIRLST* is NIL, the value of the variable DIRECTORIES is used.

> Note: If *DIRLST* is NIL, and *FILE* is not found by searching the directories on DIRECTORIES, but the root name of *FILE* has a FILEDATES property (Chapter 17) indicating that a file by that name has been loaded, then the directory indicated in the FILEDATES property is searched, too. This additional search is not done if *DIRLST* is non-NIL.

ERRORTYPELST (Chapter 14) initially contains the entry ((23 (SPELLFILE (CADR ERRORMESS) NIL NOFILESPELLFLG))), which causes SPELLFILE to be called in case of a FILE NOT FOUND error. If the variable NOFILESPELLFLG is T (its initial value), then spelling correction is not done on the file name, but DIRECTORIES is still searched. If SPELLFILE is successful, the operation will be reexecuted with the new (corrected) file name.

(**FINDFILE** *FILE NSFLG DIRLST*) [Function]

> Uses SPELLFILE to search for a file named *FILE*. If it finds one, returns its full name, with no user interaction. Specifically, it calls (SPELLFILE *FILE* T *NSFLG DIRLST*), after first performing two simple checks: If *FILE* has an explicit directory, it checks to see if a file so named exists, and if so returns that file. If *DIRLST* is NIL, it looks for *FILE* on the connected directory before calling SPELLFILE.

Listing File Directories

The function DIRECTORY allows you to conveniently specify and/or program a variety of directory operations:

(**DIRECTORY** *FILES COMMANDS DEFAULTEXT DEFAULTVERS*) [Function]

> Returns, lists, or performs arbitrary operations on all files specified by the "file group" *FILES*. A file group has the form of a regular file name, except that the character * can be used to match any number of characters, including zero, in the file name. For example, the file group A*B matches all file names beginning with the character A and ending with the character B. The file group *.DCOM matches all files with an extension of DCOM.
>
> If *FILES* does not contain an explicit extension, it is defaulted to *DEFAULTEXT*; if *FILES* does not contain an explicit version, it is defaulted to *DEFAULTVERS*. *DEFAULTEXT* and *DEFAULTVERS* themselves default to *. If the period or semicolon preceding the omitted extension or version, respectively, is present, the field is explicitly empty and no default is used. All other unspecified fields default to *. Null version is interpreted as "highest". Thus *FILES* = * or *.* or *.*;* enumerates all files on the connected directory; *FILES* = *. or *.;* enumerates all versions of files with null extension; *FILES* = *.; enumerates the highest version of files with null extension; and *FILES* = *.*; enumerates the highest version of all files. If *FILES* is NIL, it defaults to *.*;*.
>
>> Note: Some hosts/devices are not capable of supporting "highest version" in enumeration. Such hosts instead enumerate *all* versions.
>
> For each file that matches the file group *FILES*, the "file commands" in *COMMANDS* are executed in order. Some of the file commands allow aborting the command processing for a given file, effectively filtering the list of files. The interpretation of the different file commands is described below. If *COMMANDS* is NIL, it defaults to (COLLECT), which collects the matching file names in a list and returns it as the value of DIRECTORY.

The "file commands" in *COMMANDS* are interpreted as follows:

> P   Prints the file's name. For readability, DIRECTORY strips the directory from the name, printing it once as a header in front of each set of consecutive files on the same directory.
>
> PP   Prints the file's name without a version number.
>
> a string   Prints the string.

READDATE, WRITEDATE

```
CREATIONDATE, SIZE
  LENGTH, BYTESIZE
PROTECTION, AUTHOR
```
TYPE	Prints the appropriate information returned by `GETFILEINFO` (see above).
COLLECT	Adds the full name of this file to an accumulating list, which will be returned as the value of `DIRECTORY`.
COUNTSIZE	Adds the size of this file to an accumulating sum, which will be returned as the value of `DIRECTORY`.
DELETE	Deletes the file.
DELVER	If this file is not the highest version of files by its name, delete it.
PAUSE	Waits until you type any character before proceeding with the rest of the commands (good for display if you want to ponder).

The following commands are predicates to filter the list. If the predicate is not satisfied, then processing for this file is aborted and no further commands (such as those above) are executed for this file.

Note: if the `P` and `PP` commands appear in *COMMANDS* ahead of any of the filtering commands below except `PROMPT`, they are postponed until after the filters. Thus, assuming the caller has placed the attribute options after the filters as well, no printing occurs for a file that is filtered out. This is principally so that functions like `DIR` (below) can both request printing and pass arbitrary commands through to `DIRECTORY`, and have the printing happen in the appropriate place.

PROMPT *MESS*	Prompts with the yes/no question *MESS*; if user responds with `No`, abort command processing for this file.
OLDERTHAN *N*	Continue command processing if the file hasn't been referenced (read or written) in *N* days. *N* can also be a string naming an explicit date and time since which the file must not have been referenced.
NEWERTHAN *N*	Continue command processing if the file has been written within the last *N* days. *N* can also be a string naming an explicit date and time. Note that this is not quite the complement of `OLDERTHAN`, since it ignores the read date.
BY *USER*	Continue command processing if the file was last written by the given user, i.e., its `AUTHOR` attribute matches (case insensitively) *USER*.
@ *X*	*X* is either a function of one argument (*FILENAME*), or an arbitrary expression which uses the variable `FILENAME` freely. If *X* returns `NIL`, abort command processing for this file.

The following two commands apply not to any particular file, but globally to the manner in which directory information is printed.

OUT *FILE*   Directs output to *FILE*.

COLUMNS *N*   Attempts to format output in *N* columns (rather than just 1).

DIRECTORY uses the variable DIRCOMMANDS as a spelling list to correct spelling and define abbreviations and synonyms (see Chapter 20). Currently the following abbreviations are recognized:

```
           AU   =>    AUTHOR
            -   =>    PAUSE
     COLLECT?   =>    PROMPT " ? " COLLECT
           DA
         DATE   =>    CREATIONDATE
           TI   =>    WRITEDATE
          DEL   =>    DELETE
         DEL?
      DELETE?   =>    PROMPT " delete? " DELETE
          OLD   =>    OLDERTHAN 90
           PR   =>    PROTECTION
           SI   =>    SIZE
      VERBOSE   =>    AUTHOR CREATIONDATE SIZE
                       READDATE WRITEDATE
```

(**FILDIR** *FILEGROUP*)   [Function]

   Obsolete synonym of (DIRECTORY *FILEGROUP*).

(**DIR** *FILEGROUP* *COM$_1$* ... *COM$_N$*)   [NLambda NoSpread Function]

   Convenient form of DIRECTORY for use in type-in at the executive. Performs (DIRECTORY '*FILEGROUP* '(P *COM$_1$* ... *COM$_N$*)).

(**NDIR** *FILEGROUP* *COM$_1$* ... *COM$_N$*)   [NLambda NoSpread Function]

   Version of DIR that lists the file names in a multi-column format. Also, by default only lists the most recent version of files (unless *FILEGROUP* contains an explicit version).

23. STREAMS AND FILES

Medley can perform input/output operations on a large variety of physical devices, including local disk drives, floppy disk drives, the keyboard and display screen, and remote file server computers accessed over a network. While the low-level details of how all these devices perform input/output vary considerably, the Interlisp-D language provides the programmer a small, common set of abstract operations whose use is largely independent of the physical input/output medium involved— operations such as *read*, *print*, *change font*, or *go to a new line*. By merely changing the targeted I/O device, a single program can be used to produce output on the display, a file, or a printer.

The underlying data abstraction that permits this flexibility is the *stream*. A stream is a data object (an instance of the data type STREAM) that encapsulates all of the information about an input/output connection to a particular I/O device. Each of Medley's general-purpose I/O functions takes a stream as one of its arguments. The general-purpose function then performs action specific to the stream's device to carry out the requested operation. Not every device is capable of implementing every I/O operation, while some devices offer additional functionality by way of special functions for that device alone. Such restrictions and extensions are noted in the documentation of each device.

The vast majority of the streams commonly used in Medley fall into two interesting categories: the *file stream* and the *image stream*.

A file is an ordered collection of data, usually a sequence of characters or bytes, stored on a file device in a manner that allows the data to be retrieved at a later time. Floppy disks, hard disks, and remote file servers are among the devices used to store files. Files are identified by a "file name", which specifies the device on which the file resides and a name unique to a specific file on that device. Input or output to a file is performed by obtaining a stream to the file, using OPENSTREAM (see below). In addition, there are functions that manipulate the files themselves, rather than their data content.

An image stream is an output stream to a display device, such as the display screen or a printer. In addition to the standard output operations, such as print, an image stream implements a variety of graphics operations, such as drawing lines and displaying characters in multiple fonts. Unlike a file, the "content" of an image stream cannot be retrieved. Image streams are described in Chapter 26.

The creation of other kinds of streams, such as network byte-stream connections, is described in the chapters peculiar to those kinds of streams. The operations common to streams in general are described in Chapter 24. This chapter describes operations specific to file devices: how to name files, how to open streams to files, and how to manipulate files on their devices.

Opening and Closing File Streams

In order to perform input from or output to a file, it is necessary to create a stream to the file, using OPENSTREAM:

(OPENSTREAM *FILE ACCESS RECOG PARAMETERS* —)   [Function]

Opens and returns a stream for the file specified by *FILE*, a file name. *FILE* can be either a string or a symbol. The syntax and manipulation of file names is described at length in the FILENAMES section below. Incomplete file names are interpreted with respect to the connected directory (below).

RECOG specifies the recognition mode of *FILE*, as described in a later section of this chapter. If *RECOG* = NIL, it defaults according to the value of *ACCESS*.

ACCESS specifies the "access rights" to be used when opening the file, one of the following:

INPUT Only input operations are permitted on the file. The file must already exist. Starts reading at the beginning of the file. *RECOG* defaults to OLD.

OUTPUT Only output operations are permitted on the file. Starts writing at the beginning of the file, which is initially empty. While the file is open, other users or processes are unable to open the file for either input or output. *RECOG* defaults to NEW.

BOTH Both input and output operations are permitted on the file. Starts reading or writing at the beginning of the file. *RECOG* defaults to OLD/NEW. *ACCESS* = BOTH implies random accessibility (Chapter 25), and thus may not be possible for files on some devices.

APPEND Only sequential output operations are permitted on the file. Starts writing at the *end* of the file. *RECOG* defaults to OLD/NEW. *ACCESS* = APPEND may not be allowed for files on some devices.

Note: *ACCESS* = OUTPUT implies that one intends to write a new or different file, even if a version number was specified and the corresponding file already exists. Thus any previous contents of the file are discarded, and the file is empty immediately after the OPENSTREAM. If it is desired to write on an already existing file while preserving the old contents, the file must be opened for access BOTH or APPEND.

PARAMETERS is a list of pairs (*ATTRIB VALUE*), where *ATTRIB* is any file attribute that the file system is willing to allow you to set (see SETFILEINFO below). A non-list *ATTRIB* in *PARAMETERS* is treated as the pair (*ATTRIB* T). Generally speaking, attributes that belong to the permanent file (e.g., TYPE) can only be set when creating a new file, while attributes that belong only to a particular opening of a file (e.g., ENDOFSTREAMOP) can be set on any call to OPENSTREAM. Not all devices honor all attributes; those not recognized by a particular device are simply ignored.

In addition to the attributes permitted by SETFILEINFO, the following tokens are accepted by OPENSTREAM as values of *ATTRIB* in its *PARAMETERS* argument:

DON'T.CHANGE.DATE   If *VALUE* is non-NIL, the file's creation date is not changed when the file is opened. This option is meaningful only for old files being opened for access BOTH. This should be used only for specialized applications in which the caller does not want the file system to believe the file's content has been changed.

SEQUENTIAL   If *VALUE* is non-NIL, this opening of the file need support only sequential access; i.e., the caller intends never to use SETFILEPTR. For some devices, sequential access to files is much more efficient than random access. Note that the device may choose to ignore this attribute and still open the file in a manner that permits random access. Also note that this attribute does not make sense with *ACCESS* = BOTH.

If *FILE* is not recognized by the file system, OPENSTREAM causes the error FILE NOT FOUND. Ordinarily, this error is intercepted via an entry on ERRORTYPELST (Chapter 24), which causes SPELLFILE (see the Searching File Directories section of this chapter) to be called. SPELLFILE searches alternate directories and possibly attempts spelling correction on the file name. Only if SPELLFILE is unsuccessful will the FILE NOT FOUND error actually occur.

If *FILE* exists but cannot be opened, OPENSTREAM causes one of several other errors: FILE WON'T OPEN if the file is already opened for conflicting access by someone else; PROTECTION VIOLATION if the file is protected against the operation; FILE SYSTEM RESOURCES EXCEEDED if there is no more room in the file system.

(CLOSEF *FILE*)   [Function]

Closes *FILE*, and returns its full file name. Generates an error, FILE NOT OPEN, if *FILE* does not designate an open stream. After closing a stream, no further input/output operations are permitted on it.

If *FILE* is NIL, it is defaulted to the primary input stream if that is not the terminal stream, or else the primary output stream if that is not the terminal stream. If both primary input and output streams are the terminal input/output streams, CLOSEF returns NIL. If CLOSEF closes either the primary input stream or the primary output stream (either explicitly or in the *FILE* = NIL case), it resets the primary stream for that direction to be the corresponding terminal stream. See Chapter 25 for information on the primary input/output streams.

WHENCLOSE (see below) allows you to "advise" CLOSEF to perform various operations when a file is closed.

Because of buffering, the contents of a file open for output are not guaranteed to be written to the actual physical file device until CLOSEF is called. Buffered data can be

forced out to a file without closing the file by using the function `FORCEOUTPUT` (Chapter 25).

Some network file devices perform their transactions in the background. As a result, it is possible for a file to be closed by `CLOSEF` and yet not be "fully" closed for some small period of time afterward, during which time the file appears to still be busy, and cannot be opened for conflicting access by other users.

`(CLOSEF? FILE)` [Function]

Closes `FILE` if it is open, returning the value of `CLOSEF`; otherwise does nothing and returns `NIL`.

In the present implementation of Medley, all streams to files are kept, while open, in a registry of "open files". This registry does not include nameless streams, such as string streams (see below), display streams (Chapter 28), and the terminal input and output streams; nor streams explicitly hidden from you, such as dribble streams (Chapter 30). This registry may not persist in future implementations of Medley, but at the present time it is accessible by the following two functions:

`(OPENP FILE ACCESS)` [Function]

`ACCESS` is an access mode for a stream opening (one of `INPUT`, `OUTPUT`, `BOTH`, or `APPEND`), or `NIL`, meaning any access.

If `FILE` is a stream, returns its full name if it is open for the specified access, else `NIL`.

If `FILE` is a file name (a symbol), `FILE` is processed according to the rules of file recognition (see below). If a stream open to a file by that name is registered and open for the specified access, then the file's full name is returned. If the file name is not recognized, or no stream is open to the file with the specified access, `NIL` is returned.

If `FILE` is `NIL`, returns a list of the full names of all registered streams that are open for the specified access.

`(CLOSEALL ALLFLG)` [Function]

Closes all streams in the value of `(OPENP)`. Returns a list of the files closed.

`WHENCLOSE` (see below) allows certain files to be "protected" from `CLOSEALL`. If `ALLFLG` is `T`, all files, including those protected by `WHENCLOSE`, are closed.

File Names

A file name in Medley is a string or symbol whose characters specify a "path" to the actual file: on what host or device the file resides, in which directory, and so forth. Because Medley supports a variety of non-local file devices, parts of the path could be very device-dependent. However, it is desirable for programs to be able to manipulate file names in a device-independent manner. To this end, Medley specifies a uniform file name syntax over all devices; the functions that perform the

actual file manipulation for a particular device are responsible for any translation to that device's naming conventions.

A file name is composed of a collection of *fields*, some of which have specific semantic interpretations. The functions described below refer to each field by a *field name*, a literal atom from among the following: HOST, DEVICE, DIRECTORY, NAME, EXTENSION, and VERSION. The standard syntax for a file name that contains all of those fields is {HOST}DEVICE:<DIRECTORY>NAME.EXTENSION;VERSION. Some host's file systems do not use all of those fields in their file names.

<table>
<tr><td>HOST</td><td>Specifies the host whose file system contains the file. In the case of local file devices, the "host" is the name of the device, e.g., DSK or FLOPPY.</td></tr>
<tr><td>DEVICE</td><td>Specifies, for those hosts that divide their file system's name space among mutiple physical devices, the device or logical structure on which the file resides. This should not be confused with Medley's abstract "file device", which denotes either a host or a local physical device and is specified by the HOST field.</td></tr>
<tr><td>DIRECTORY</td><td>Specifies the "directory" containing the file. A directory usually is a grouping of a possibly large set of loosely related files, e.g., the personal files of a particular user, or the files belonging to some project. The DIRECTORY field usually consists of a principal directory and zero or more subdirectories that together describe a path through a file system's hierarchy. Each subdirectory name is set off from the previous directory or subdirectory by the character ">"; e.g., "LISP>LIBRARY>NEW".</td></tr>
<tr><td>NAME</td><td>This field carries no specific meaning, but generally names a set of files thought of as being different renditions of the "same" abstract file.</td></tr>
<tr><td>EXTENSION</td><td>This field also carries no specific meaning, but generally distinguishes the form of files having the same name. Most files systems have some "conventional" extensions that denote something about the content of the file. For example, in Medley, the extension DCOM standardly denotes a file containing compiled function definitions.</td></tr>
<tr><td>VERSION</td><td>A number used to distinguish the versions or "generations" of the files having a common name and extension. The version number is incremented each time a new file by the same name is created.</td></tr>
</table>

Most functions that take as input "a directory" accept either a directory name (the contents of the DIRECTORY field of a file name) or a "full" directory specification—a file name fragment consisting of

only the fields HOST, DEVICE, and DIRECTORY. In particular, the "connected directory" (see below) consists, in general, of all three fields.

For convenience in dealing with certain operating systems, Medley also recognizes [] and () as host delimiters (synonymous with {}), and / as a directory delimiter (synonymous with < at the beginning of a directory specification and **>** to terminate directory or subdirectory specification). For example, a file on a Unix file server UNX with the name /usr/foo/bar/stuff.tedit, whose DIRECTORY field is thus usr/foo/bar, could be specified as {UNX}/usr/foo/bar/stuff.tedit, or (UNX)<usr/foo/bar>stuff.tedit, or several other variations. Note that when using [] or () as host delimiters, they usually must be escaped with the reader's % escape character if the file name is expressed as a symbol rather than a string.

Different hosts have different requirements regarding which characters are valid in file names. From Medley's point of view, any characters are valid. However, in order to be able to parse a file name into its component fields, it is necessary that those characters that are conventionally used as file name delimiters be quoted when they appear inside of fields where there could be ambiguity. The file name quoting character is "'" (single quote). Thus, the following characters must be quoted when not used as delimeters: :, >, ;, /, and ' itself. The character . (period) need only be quoted if it is to be considered a part of the EXTENSION field. The characters },], and) need only be quoted in a file name when the host field of the name is introduced by {, [, and (, respectively. The characters {, [, (, and < need only be quoted if they appear as the first character of a file name fragment, where they would otherwise be assumed to introduce the HOST or DIRECTORY fields.

The following functions are the standard way to manipulate file names in Interlisp. Their operation is purely syntactic—they perform no file system operations themselves.

> (UNPACKFILENAME.STRING *FILENAME*)   [Function]

> > Parses , returning a list in property list format of alternating field names and field contents. The field contents are returned as strings. If is a stream, its full name is used.

> > Only those fields actually present in are returned. A field is considered present if its delimiting punctuation (in the case of EXTENSION and VERSION, the preceding period or semicolon, respectively) is present, even if the field itself is empty. Empty fields are denoted by "" (the empty string).

> > Examples:

> > > ```
> > > (UNPACKFILENAME.STRING "FOO.BAR") =>
> > >    (NAME "FOO" EXTENSION "BAR")
> > > 
> > > (UNPACKFILENAME.STRING "FOO.;2") =>
> > >    (NAME "FOO" EXTENSION "" VERSION "2")
> > > 
> > > (UNPACKFILENAME.STRING "FOO;") =>
> > >    (NAME "FOO" VERSION "")
> > > 
> > > (UNPACKFILENAME.STRING
> > >   "{ERIS}<LISP>CURRENT>IMTRAN.DCOM;21")
> > >  => (HOST "ERIS" DIRECTORY "LISP>CURRENT"
> > >        NAME "IMTRAN" EXTENSION "DCOM"
> > >        VERSION "21")
> > > ```

(UNPACKFILENAME *FILE*)   [Function]

> Old version of UNPACKFILENAME.STRING that returns the field values as atoms, rather than as strings. UNPACKFILENAME.STRING is now considered the "correct" way of unpacking file names, because it does not lose information when the contents of a field are numeric. For example,
>
> ```
> (UNPACKFILENAME 'STUFF.TXT) =>
>    (NAME STUFF EXTENSION TXT)
> ```
>
> but
>
> ```
> (UNPACKFILENAME 'STUFF.029) =>
>    (NAME STUFF EXTENSION 29)
> ```
>
> Explicitly omitted fields are denoted by the atom NIL, rather than the empty string.
>
> > Note: Both UNPACKFILENAME and UNPACKFILENAME.STRING leave the trailing colon on the device field, so that the Tenex device NIL: can be distinguished from the absence of a device. Although UNPACKFILENAME.STRING is capable of making the distinction, it retains this behavior for backward compatibility. Thus,
> >
> > ```
> > (UNPACKFILENAME.STRING '{TOAST}DSK:FOO) =>
> >    (HOST "TOAST" DEVICE "DSK:" NAME "FOO")
> > ```

(FILENAMEFIELD *FILENAME FIELDNAME*)   [Function]

> Returns, as an atom, the contents of the *FIELDNAME* field of *FILENAME*. If *FILENAME* is a stream, its full name is used.

(PACKFILENAME.STRING *FIELD₁ CONTENTS₁ ... FIELDₙ CONTENTSₙ*)   [NoSpread Function]

> Takes a sequence of alternating field names and field contents (atoms or strings), and returns the corresponding file name, as a string.
>
> If PACKFILENAME.STRING is given a single argument, it is interpreted as a list of alternating field names and field contents. Thus PACKFILENAME.STRING and UNPACKFILENAME.STRING operate as inverses.
>
> If the same field name is given twice, the *first* occurrence is used.
>
> The contents of the field name DIRECTORY may be either a directory name or a full directory specification as described above.
>
> PACKFILENAME.STRING also accepts the "field name" BODY to mean that its contents should itself be unpacked and spliced into the argument list at that point. This feature, in conjunction with the rule that fields early in the argument list override later duplicates, is useful for altering existing file names. For example, to provide a default field, place BODY first in the argument list, then the default fields. To override a field, place the new fields first and BODY last.
>
> If the value of the BODY field is a stream, its full name is used.

Examples:

```
(PACKFILENAME.STRING 'DIRECTORY "LISP"
   'NAME "NET")
      =>  "<LISP>NET"
(PACKFILENAME.STRING 'NAME "NET"
   'DIRECTORY "{DSK}<LISPFILES>")
      => "{DSK}<LISPFILES>NET"
(PACKFILENAME.STRING 'DIRECTORY "{DSK}"
   'BODY "{TOAST}<FOO>BAR")
      => "{DSK}BAR"
(PACKFILENAME.STRING 'DIRECTORY "FRED"
   'BODY "{TOAST}<FOO>BAR")
      => "{TOAST}<FRED>BAR"
(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR"
   'DIRECTORY "FRED")
      => "{TOAST}<FOO>BAR"
(PACKFILENAME.STRING 'VERSION NIL
   'BODY "{TOAST}<FOO>BAR.DCOM;2")
      => "{TOAST}<FOO>BAR.DCOM"
(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR.DCOM"
   'VERSION 1)
      => "{TOAST}<FOO>BAR.DCOM;1"
(PACKFILENAME.STRING 'BODY "{TOAST}<FOO>BAR.DCOM;"
   'VERSION 1)
      => "{TOAST}<FOO>BAR.DCOM;"
(PACKFILENAME.STRING 'BODY "BAR.;1"
   'EXTENSION "DCOM")
      => "BAR.;1"
(PACKFILENAME.STRING 'BODY "BAR;1"
   'EXTENSION "DCOM")
      => "BAR.DCOM;1"
```

In the last two examples, note that in one case the extension is explicitly present in the body (as indicated by the preceding period), while in the other there is no indication of an extension, so the default is used.

(PACKFILENAME $FIELD_1$ $CONTENTS_1$ ... $FIELD_N$ $CONTENTS_N$)   [NoSpread Function]

The same as PACKFILENAME.STRING, except that it returns the file name as a symbol, instead of a string.

Incomplete File Names

In general, it is not necessary to pass a complete file name (one containing all the fields listed above) to functions that take a file name as argument. Interlisp supplies suitable defaults for certain fields, as described below. Functions that return names of actual files, however, always return the fully specified name.

If the version field is omitted from a file name, Interlisp performs version recognition, as described below.

If the host, device and/or directory field are omitted from a file name, Interlisp defaults them with respect to the currently connected directory. The connected directory is changed by calling the function CNDIR or using the programmer's assistant command CONN.

Defaults are added to the partially specified name "left to right" until a host, device or directory field is encountered. Thus, if the connected directory is {TWENTY}PS:<FRED>, then

>   BAR.DCOM means
>   {TWENTY}PS:<FRED>BAR.DCOM
>
>   <GRANOLA>BAR.DCOM means
>   {TWENTY}PS:<GRANOLA>BAR.DCOM
>
>   MTA0:<GRANOLA>BAR.DCOM means
>   {TWENTY}MTA0:<GRANOLA>BAR.DCOM
>
>   {THIRTY}<GRANOLA>BAR.DCOM means
>   {THIRTY}<GRANOLA>BAR.DCOM

In addition, if the partially specified name contains a subdirectory, but no principal directory, then the subdirectory is appended to the connected directory. For example,

>   ISO>BAR.DCOM means
>   {TWENTY}PS:<FRED>ISO>BAR.DCOM

Or, if the connected directory is the Unix directory {UNX}/usr/fred/, then iso/bar.dcom means {UNX}/usr/fred/iso/bar.dcom, but /other/bar.dcom means {UNX}/other/bar.dcom.

(CNDIR *HOST/DIR*)   [Function]

> Connects to the directory *HOST/DIR*, which can either be a directory name or a full directory specification including host and/or device. If the specification includes just a host, and the host supports directories, the directory is defaulted to the value of (USERNAME); if the host is omitted, connection is made to another directory on the same host as before. If *HOST/DIR* is NIL, connects to the value of LOGINHOST/DIR.

> CNDIR returns the full name of the now-connected directory. Causes an error, Non-existent directory, if *HOST/DIR* is not recognized as a valid directory.

> Note that CNDIR does not necessarily require or provide any directory access privileges. Access privileges are checked when a file is opened.

CONN *HOST/DIR*   [Prog. Asst. Command]

> Convenient command form of CNDIR for use at the executive. Connects to *HOST/DIR*, or to the value of LOGINHOST/DIR if *HOST/DIR* is omitted. This command is undoable—undoing it causes the system to connect to the previously connected directory.

LOGINHOST/DIR   [Variable]

> CONN with no argument connects to the value of the variable LOGINHOST/DIR, initially {DSK}, but usually reset in your greeting file (Chapter 12).

(DIRECTORYNAME *DIRNAME STRPTR*)   [Function]

> If *DIRNAME* is T, returns the full specification of the currently connected directory. If *DIRNAME* is NIL, returns the "login" directory specification (the value of LOGINHOST/DIR). For any other value of *DIRNAME*, returns a full directory specification if *DIRNAME* designates an existing directory (satisfies DIRECTORYNAMEP), otherwise NIL.

> If *STRPTR* is T, the value is returned as an atom, otherwise it is returned as a string.

(DIRECTORYNAMEP *DIRNAME HOSTNAME*)   [Function]

> Returns T if *DIRNAME* is recognized as a valid directory on host *HOSTNAME*, or on the host of the currently connected directory if *HOSTNAME* is NIL. *DIRNAME* may be either a directory name or a full directory specification containing host and/or device as well.

> If *DIRNAME* includes subdirectories, this function may or may not pass judgment on their validity. Some hosts support "true" subdirectories, distinct entities manipulable by the file system, while others only provide them as a syntactic convenience.

(HOSTNAMEP *NAME*)   [Function]

> Returns T if *NAME* is recognized as a valid host or file device name at the moment HOSTNAMEP is called.

Version Recognition

Most of the file devices in Interlisp support file version numbers. That is, it is possible to have several files of the exact same name, differing only in their VERSION field, which is incremented for each new "version" of the file that is created. When a file name lacking a version number is presented to the file system, it is necessary to determine which version number is intended. This process is known as *version recognition.*

When OPENSTREAM opens a file for input and no version number is given, the highest existing version number is used. Similarly, when a file is opened for output and no version number is given, a new file is created with a version number one higher than the highest one currently in use with that file name. The version number defaulting for OPENSTREAM can be changed by specifying a different value for its *RECOG* argument, as described under FULLNAME, below.

Other functions that accept file names as arguments generally perform the default version recognition, which is newest version for existing files, or a new version if using the file name to create a new file. The one exception is DELFILE, which defaults to the oldest existing version of the file.

The functions below can be used to perform version recognition without actually calling `OPENSTREAM` to open the file. Note that these functions only tell the truth about the moment at which they are called, and thus cannot in general be used to anticipate the name of the file opened by a comparable `OPENSTREAM`. They are sometimes, however, helpful hints.

(`FULLNAME` *X RECOG*)   [Function]

> If *X* is an open stream, simply returns the full file name of the stream. Otherwise, if *X* is a file name given as a string or symbol, performs version recognition, as follows:
>
> If *X* is recognized in the recognition mode specified by *RECOG* as an abbreviation for some file, returns the file's full name, otherwise `NIL`. *RECOG* is one of the following:

`OLD`	Choose the newest existing version of the file. Return `NIL` if no file named *X* exists.
`OLDEST`	Choose the oldest existing version of the file. Return `NIL` if no file named *X* exists.
`NEW`	Choose a new (not yet existing) version of the file. That is, if versions of *X* already exist, then choose a version number one higher than highest existing version; else choose version 1. For some file systems, `FULLNAME` returns `NIL` if you do not have the access rights necessary for creating a new file named *X*.
`OLD/NEW`	Try `OLD`, then `NEW`. That is, choose the newest existing version of the file, if any; else choose version 1. This usually only makes sense if you are intending to open *X* for access `BOTH`.

> > *RECOG* = `NIL` defaults to `OLD`. For all other values of *RECOG*, generates an error `ILLEGAL ARG`.
> >
> > If *X* already contains a version number, the *RECOG* argument will never change it. In particular, *RECOG* = `NEW` does not require that the file actually be new. For example, (`FULLNAME` `'FOO.;2` `'NEW`) may return {`ERIS`}<`LISP`>`FOO.;2` if that file already exists, even though (`FULLNAME` `'FOO` `'NEW`) would default the version to a new number, perhaps returning {`ERIS`}<`LISP`>`FOO.;5`.

(`INFILEP` *FILE*)   [Function]

> Equivalent to (`FULLNAME` *FILE* `'OLD`). That is, returns the full file name of the newest version of *FILE* if *FILE* is recognized as specifying the name of an existing file that could potentially be opened for input, `NIL` otherwise.

    `(OUTFILEP FILE)`   [Function]

        Equivalent to `(FULLNAME FILE 'NEW)`.

Note that `INFILEP`, `OUTFILEP` and `FULLNAME` do not open any files; they are pure predicates. In general they are also only hints, as they do not necessarily imply that the caller has access rights to the file. For example, `INFILEP` might return non-`NIL`, but `OPENSTREAM` might fail for the same file because the file is read-protected against you, or the file happens to be open for output by another user at the time. Similarly, `OUTFILEP` could return non-`NIL`, but `OPENSTREAM` could fail with a `FILE SYSTEM RESOURCES EXCEEDED` error.

Note also that in a shared file system, such as a remote file server, intervening file operations by another user could contradict the information returned by recognition. For example, a file that was `INFILEP` might be deleted, or between an `OUTFILEP` and the subsequent `OPENSTREAM`, another user might create a new version or delete the highest version, causing `OPENSTREAM` to open a different version of the file than the one returned by `OUTFILEP`. In addition, some file servers do not well support recognition of files in output context. Thus, in general, the "truth" about a file can only be obtained by actually opening the file; creators of files should rely on the name of the stream opened by `OPENSTREAM`, not the value returned from these recognition functions. In particular, for the reasons described earlier, programmers are discouraged from using `OUTFILEP` or `(FULLNAME NAME 'NEW)`.

Using File Names Instead of Streams

In earlier implementations of Interlisp, from the days of Interlisp-10 onward, the "handle" used to refer to an open file was not a stream, but rather the file's full name, represented as a symbol. When the file name was passed to any I/O function, it was mapped to a stream by looking it up in a list of open files. This scheme was sometimes convenient for typing in file commands at the executive, but was very poor for serious programming in two major ways. First, the mapping from file name to stream on every input/output operation is inefficient. Second, and more importantly, using the file name as the handle on an open stream means that it is not possible to have more than one stream open on a given file at once.

As of this writing, Medley is in a transition period, where it still supports the use of symbol file names as synonymous with open streams, but this use is not recommended. The remainder of this section discusses this usage of file names for the benefit of those reading older programs and wishing to convert them as necessary to work properly when this compatibility feature is removed.

File Name Efficiency Considerations

It is possible for a program to be seriously inefficient using a file name as a stream if the program is not using the file's full name, the name returned by `OPENFILE` (below). Any time that an input/output function is called with a file name other than the full file name, Interlisp must perform recognition on the partial file name in order to determine which open file is intended. Thus if repeated operations are to be performed, it is considerably more efficient to use the full file name

returned from OPENFILE than to repeatedly use the possibly incomplete name that was used to open the file.

There is a more subtle problem with partial file names, in that recognition is performed on your entire directory, not just the open files. It is possible for a file name that was previously recognized to denote one file to suddenly denote a different file. For example, suppose a program performs (INFILE 'FOO), opening FOO.;1, and reads several expressions from FOO. Then you interrupt the program, create a FOO.;2 and resume the program (or a user at another workstation creates a FOO.;2). Now a call to READ giving it FOO as its *FILE* argument will generate a FILE NOT OPEN error, because FOO will be recognized as FOO.;2.

Obsolete File Opening Functions

The following functions are now considered obsolete, but are provided for backwards compatibility:

(OPENFILE *FILE ACCESS RECOG PARAMETERS*) [Function]

> Opens *FILE* with access rights as specified by *ACCESS*, and recognition mode *RECOG*, and returns the full name of the resulting stream. Equivalent to (FULLNAME (OPENSTREAM *FILE ACCESS RECOG PARAMETERS*)).

(INFILE *FILE*) [Function]

> Opens *FILE* for input, and sets it as the primary input stream. Equivalent to (INPUT (OPENSTREAM *FILE* 'INPUT 'OLD))

(OUTFILE *FILE*) [Function]

> Opens *FILE* for output, and sets it as the primary output stream. Equivalent to (OUTPUT (OPENSTREAM *FILE* 'OUTPUT 'NEW)).

(IOFILE *FILE*) [Function]

> Equivalent to (OPENFILE *FILE* 'BOTH 'OLD); opens *FILE* for both input and output. Does not affect the primary input or output stream.

Converting Old Programs

At some point in the future, the Medley file system will change so that each call to OPENSTREAM returns a distinct stream, even if a stream is already open to the specified file. This change is required in order to deal rationally with files in a multiprocessing environment.

This change will of necessity produce the following incompatibilities:

1. The functions OPENFILE, INPUT, and OUTPUT will return a STREAM, not a full file name. To make this less confusing in interactive situations, STREAMs will have a print format that reveals the underlying file's actual name,

2. A greater penalty will ensue for passing as the *FILE* argument to I/O operations anything other than the object returned from OPENFILE. Passing the file's name will be significantly slower than passing the stream (even when passing the "full" file name), and in the case where there is more than one stream open on the file it might even act on the wrong one.

3. OPENP will return NIL when passed the name of a file rather than a stream (the value of OPENFILE or OPENSTREAM).

Users should consider the following advice when writing new programs and editing existing programs, in order that they will continue to operate well when this change is made:

Because of the efficiency and ambiguity considerations described earlier, users have long been encouraged to use only full file names as *FILE* arguments to I/O operations. The "proper" way to have done this was to bind a variable to the value returned from OPENFILE and pass that variable to all I/O operations; such code will continue to work. A less proper way to obtain the full file name, but one which has to date not incurred any obvious penalty, is that which binds a variable to the result of an INFILEP and passes that to OPENFILE and all I/O operations. This has worked because INFILEP and OPENFILE both return a full file name, an invalid assumption in this future world. Such code should be changed to pass around the value of the OPENFILE, not the INFILEP.

Code that calls OPENP to test whether a possibly incomplete file name is already open should be recoded to pass to OPENP only the value returned from OPENFILE or OPENSTREAM.

Code that uses ordinary string functions to manipulate file names, and in particular the value returned from OPENFILE, should be changed to use the the functions UNPACKFILENAME.STRING and PACKFILENAME.STRING. Those functions work both on file names (strings) and streams (coercing the stream to the name of its file).

Code that tests the value of OUTPUT for equality to some known file name or T should be examined carefully and, if possible, recoded.

To see more directly the effects of passing around STREAMs instead of file names, replace your calls to OPENFILE with calls to OPENSTREAM. OPENSTREAM is called in exactly the same way, but returns a STREAM. Streams can be passed to READ, PRINT, CLOSEF, etc just as the file's full name can be currently, but using them is more efficient. The function FULLNAME, when applied to a stream, returns its full file name.

Using Files with Processes

Because Medley does not yet support multiple streams per file, problems can arise if different processes attempt to access the same file. You have to be careful not to have two processes manipulating the same file at the same time, since the two processes will be sharing a single input

stream and file pointer. For example, it will not work to have one process `TCOMPL` a file while another process is running `LISTFILES` on it.

File Attributes

Any file has a number of "file attributes", such as the read date, protection, and bytesize. The exact attributes that a file can have is dependent on the file device. The functions `GETFILEINFO` and `SETFILEINFO` allow you to conveniently access file attributes:

(`GETFILEINFO` *FILE ATTRIB*)   [Function]

> Returns the current setting of the *ATTRIB* attribute of *FILE*.

(`SETFILEINFO` *FILE ATTRIB VALUE*)   [Function]

> Sets the attribute *ATTRIB* of *FILE* to be *VALUE*. `SETFILEINFO` returns `T` if it is able to change the attribute *ATTRIB*, and `NIL` if unsuccessful, either because the file device does not recognize *ATTRIB* or because the file device does not permit the attribute to be modified.

The *FILE* argument to `GETFILEINFO` and `SETFILEINFO` can be an open stream (or an argument designating an open stream, see Chapter 25), or the name of a closed file. `SETFILEINFO` in general requires write access to the file.

The attributes recognized by `GETFILEINFO` and `SETFILEINFO` fall into two categories: *permanent* attributes, which are properties of the file, and *temporary* attributes, which are properties only of an open stream to the file. The temporary attributes are only recognized when *FILE* designates an open stream; the permanent attributes are usually equally accessible for open and closed files. However, some devices are willing to change the value of certain attributes of an open stream only when specified in the *PARAMETERS* argument to `OPENSTREAM` (see above), not on a later call to `SETFILEINFO`.

The following are currently recognized as permanent attributes of a file:

BYTESIZE	The byte size of the file. Medley currently only supports byte size 8.
LENGTH	The number of bytes in the file. Alternatively, the byte position of the end-of-file. Like (`GETEOFPTR` *FILE*), but *FILE* does not have to be open.
SIZE	The size of *FILE* in pages.
CREATIONDATE	The date and time, as a string, that the content of *FILE* was "created". The creation date changes whenever the content of the file is modified, but remains unchanged when a file is transported,

	unmodified, across file systems. Specifically, `COPYFILE` and `RENAMEFILE` (see below) preserve the file's creation date. Note that this is different from the concept of "creation date" used by some operating systems (e.g., Tops20).
`WRITEDATE`	The date and time, as a string, that the content of *FILE* was last written to this particular file system. When a file is copied, its creation date does not change, but its write date becomes the time at which the copy is made.
`READDATE`	The date and time, as a string, that *FILE* was last read, or `NIL` if it has never been read.

`ICREATIONDATE`
`IWRITEDATE`
`IREADDATE`
	The `CREATIONDATE`, `WRITEDATE` and `READDATE`, respectively, in integer form, as `IDATE` (Chapter 12) would return. This form is useful for comparing dates.

`AUTHOR`	The name of the user who last wrote the file.
`TYPE`	The "type" of the file, some indication of the nature of the file's content. The "types" of files allowed depends on the file device. Most devices recognize the symbol `TEXT` to mean that the file contains just characters, or `BINARY` to mean that the file contains arbitrary data.

Some devices support a wider range of file types that distinguish among the various sorts of files one might create whose content is "binary". All devices interpret any value of `TYPE` that they do not support to be `BINARY`. Thus, `GETFILEINFO` may return the more general value `BINARY` instead of the original type that was passed to `SETFILEINFO` or `OPENSTREAM`. Similarly, `COPYFILE`, while attempting to preserve the `TYPE` of the file it is copying, may turn, say, an `INTERPRESS` file into a mere `BINARY` file.

The way in which some file devices (e.g., Xerox file servers) support a wide range of file types is by representing the type as an integer, whose interpretation is known by the client. The variable `FILING.TYPES` is used to associate symbolic types with numbers for these devices. This list initially contains some of the well-known assignments of

type name to number; you can add additional elements to handle any private file types. For example, suppose there existed an NS file type `MAZEFILE` with numeric value 5678. You could add the element `(MAZEFILE 5678)` to `FILING.TYPES` and then use `MAZEFILE` as a value for the `TYPE` attribute to `SETFILEINFO` or `OPENSTREAM`. Other devices are, of course, free to store `TYPE` attributes in whatever manner they wish, be it numeric or symbolic. `FILING.TYPES` is merely considered the official registry for Xerox file types.

For most file devices, the `TYPE` of a newly created file, if not specified in the *PARAMETERS* argument to `OPENSTREAM`, defaults to the value of `DEFAULTFILETYPE`, initially `TEXT`.

The following are currently recognized as temporary attributes of an open stream:

ACCESS
The current access rights of the stream (see the beginning of this chapter). Can be one of `INPUT`, `OUTPUT`, `BOTH`, `APPEND`; or `NIL` if the stream is not open.

ENDOFSTREAMOP
The action to be taken when a stream is at "end of file" and an attempt is made to take input from it. The value of this attribute is a function of one argument, the stream. The function can examine the stream and its calling context and take any action it wishes. If the function returns normally, its should return either `T`, meaning to try the input operation again, or the byte that `BIN` would have returned had there been more bytes to read. Ordinarily, one should not let the `ENDOFSTREAMOP` function return unless one is only performing binary input from the file, since there is no way in general of knowing in what state the reader was at the time the end of file occurred, and hence how it will interpret a single byte returned to it.

The default `ENDOFSTREAMOP` is a system function that causes the error `END OF FILE`. The behavior of that error can be further modified for a particular stream by using the `EOF` option of `WHENCLOSE` (see below).

EOL
The end-of-line convention for the stream. This can be `CR`, `LF`, or `CRLF`, indicating with what byte or sequence of bytes the "End Of Line" character is

represented on the stream. On input, that sequence of bytes on the stream is read as `(CHARCODE EOL)` by `READCCODE` or the string reader. On output, `(TERPRI)` and `(PRINTCCODE (CHARCODE EOL))` cause that sequence of bytes to be placed on the stream.

The end of line convention is usually not apparent to you. The file system is usually aware of the convention used by a particular remote operating system, and sets this attribute accordingly. If you believe a file actually is stored with a different convention than the default, it is possible to modify the default behavior by including the `EOL` attribute in the *PARAMETERS* argument to `OPENSTREAM`.

BUFFERS Value is the number of 512-byte buffers that the stream maintains at one time. This attribute is only used by certain random-access devices (currently, the local disk, floppy, and Leaf servers); all others ignore it.

Streams open to files generally maintain some portion of the file buffered in memory, so that each call to an I/O function does not require accessing the actual file on disk or a file server. For files being read or written sequentially, not much buffer space is needed, since once a byte is read or written, it will never need to be seen again. In the case of random access streams, buffering is more complicated, since a program may jump around in the file, using `SETFILEPTR` (Chapter 25). In this case, the more buffer space the stream has, the more likely it is that after a `SETFILEPTR` to a place in the file that has already been accessed, the stream still has that part of the file buffered and need not go out to the device again. This benefit must, of course, be traded off against the amount of memory consumed by the buffers.

Closing and Reopening Files

The function `WHENCLOSE` permits you to associate certain operations with open streams that govern how and when the stream will be closed. You can specify that certain functions will be executed before `CLOSEF` closes the stream and/or after `CLOSEF` closes the stream. You can make a particular stream be invisible to `CLOSEALL`, so that it will remain open across user invocations of `CLOSEALL`.

(WHENCLOSE *FILE PROP$_1$ VAL$_1$ ... PROP$_N$ VAL$_N$*)   [NoSpread Function]

FILE must designate an open stream other than T (NIL defaults to the primary input stream, if other than T, or primary output stream if other than T). The remaining arguments specify properties to be associated with the full name of *FILE*. WHENCLOSE returns the full name of *FILE* as its value.

WHENCLOSE recognizes the following property names:

BEFORE	*VAL* is a function that CLOSEF will apply to the stream just before it is closed. This might be used, for example, to copy information about the file from an in-core data structure to the file just before it is closed.
AFTER	*VAL* is a function that CLOSEF will apply to the stream just after it is closed. This capability permits in-core data structures that know about the stream to be cleaned up when the stream is closed.
CLOSEALL	*VAL* is either YES or NO and determines whether *FILE* will be closed by CLOSEALL (YES) or whether CLOSEALL will ignore it (NO). CLOSEALL uses CLOSEF, so that any AFTER functions will be executed if the stream is in fact closed. Files are initialized with CLOSEALL set to YES.
EOF	*VAL* is a function that will be applied to the stream when an end-of-file error occurs, and the ERRORTYPELST entry for that error, if any, returns NIL. The function can examine the context of the error, and can decide whether to close the stream, RETFROM some function, or perform some other computation. If the function supplied returns normally (i.e., does not RETFROM some function), the normal error machinery will be invoked.

The default EOF behavior, unless overridden by this WHENCLOSE option, is to call the value of DEFAULTEOFCLOSE (below).

For some applications, the ENDOFSTREAMOP attribute (see above) is a more useful way to intercept the end-of-file error. The ENDOFSTREAMOP attribute comes into effect before the error machinery is ever activated.

Multiple AFTER and BEFORE functions may be associated with a file; they are executed in sequence with the most recently associated function executed

first. The `CLOSEALL` and `EOF` values, however, will override earlier values, so only the last value specified will have an effect.

`DEFAULTEOFCLOSE`   [Variable]

Value is the name of a function that is called by default when an end of file error occurs and no `EOF` option has been specified for the stream by `WHENCLOSE`. The initial value of `DEFAULTEOFCLOSE` is `NILL`, meaning take no special action (go ahead and cause the error). Setting it to `CLOSEF` would cause the stream to be closed before the rest of the error machinery is invoked.

Local Hard Disk Device

Warning: This section describes the Medley functions that control the local hard disk drive available on some computers. All of these functions may not work on all computers running Medley. For more information on using the local hard disk facilities, see the users guide for your computer.

This section describes the local file system currently supported on the Xerox 1108 and 1186 computers. The Xerox 1132 supports a simpler local file system. The functions below are no-ops on the Xerox 1132, except for `DISKPARTITION` (which returns a disk partition number), and `DISKFREEPAGES`. On the Xerox 1132, different numbered partitions are referenced by using devices such as {DSK1}, {DSK2}, etc. {DSK} always refers to the disk partition that Interlisp is running on. The 1132 local file system does not support the use of directories.

The hard disk used with the Xerox 1108 or 1186 may be partitioned into a number of named "logical volumes." Logical volumes may be used to hold the Interlisp virtual memory file (see Chapter 12), or Interlisp files. For information on intializing and partitioning the hard disk, see the users guide for your computer. In order to store Interlisp files on a logical volume, it is necessary to create a lisp file directory on that volume (see `CREATEDSKDIRECTORY`, below).

So long as there exists a logical volume with a Lisp directory on it, files on this volume can be accessed by using the file device called {DSK}. Medley can be used to read, write, and otherwise interact with files on local disk disks through standard Interlisp input/output functions. All I/O functions such as `LOAD`, `OPENSTREAM`, `READ`, `PRINT`, `GETFILEINFO`, `COPYFILE`, etc., work with files on the local disk.

If you do not have a logical volume with a Lisp directory on it, Interlisp emulates the {DSK} device by a core device, a file device whose backing store is entirely within the Lisp virtual memory. However, this is not recommended because the core device only provides limited scratch space, and since the core device is contained in virtual memory, it (and the files stored on it) will be erased when the virtual memory file is reloaded.

Each logical volume with a Lisp directory on it serves as a directory of the device {DSK}. Files are referred to by forms such as

```
{DSK}<VOLUMENAME>FILENAME
```

Thus, the file INIT.LISP on the volume LISPFILES would be called {DSK}<LISPFILES>INIT.LISP.

Subdirectories within a logical volume are supported, using the > character in file names to delimit subdirectory names. For example, the file name {DSK}<LISPFILES>DOC>DESIGN.TEDIT designates the file names DESIGN.TEDIT on the subdirectory DOC on the logical volume LISPFILES.

If a logical volume name is not specified, it defaults in an unusual but simple way: the logical volume defaults to the next logical volume that has a lisp file directory on it including or after the volume containing the currently running virtual memory. For example, if the local disk has the logical volumes LISP, TEMP, and LISPFILES, the LISP volume contains the running virtual memory, and only the LISP volume has a Lisp file directory on it, then {DSK}INIT.LISP refers to the file {DSK}<LispFiles>INIT.LISP. All the functions below default logical volume names in a similar way, except for those such as CREATEDSKDIRECTORY. To determine the current default lisp file directory, evaluate (DIRECTORYNAME '{DSK}).

(CREATEDSKDIRECTORY *VOLUMENAME*) [Function]

> Creates a Lisp file directory on the logical volume *VOLUMENAME*, and returns the name of the directory created. It is only necessary to create a Lisp file directory the first time the logical volume is used. After that, the system automatically recognizes and opens access to the logical volumes that have Lisp file directories on them.

(PURGEDSKDIRECTORY *VOLUMENAME*) [Function]

> Erases all Lisp files on the volume *VOLUMENAME*, and deletes the Lisp file directory.

(LISPDIRECTORYP *VOLUMENAME*) [Function]

> Returns T if the logical volume *VOLUMENAME* has a lisp file directory on it.

(VOLUMES) [Function]

> Returns a list of the names of all of the logical volumes on the local hard disk (whether they have lisp file directories or not).

(VOLUMESIZE *VOLUMENAME*) [Function]

> Returns the total size of the logical volume *VOLUMENAME* in disk pages.

(DISKFREEPAGES *VOLUMENAME*) [Function]

> Returns the total number of free disk pages left on the logical volume *VOLUMENAME*.

(DISKPARTITION) [Function]

> Returns the name of the logical volume containing the virtual memory file that Interlisp is currently running in (see Chapter 12).

(DSKDISPLAY *NEWSTATE*)   [Function]

> Controls a display window that displays information about the logical volumes on the local hard disk (logical volume names, sizes, free pages, etc.). DSKDISPLAY opens or closes this display window depending on the value of *NEWSTATE* (one of ON, OFF, or CLOSED), and returns the previous state of the display window.
>
> If *NEWSTATE* is ON, the display window is opened, and it is automatically updated whenever the file system state changes (this can slow file operations significantly). If *NEWSTATE* is OFF, the display window is opened, but it is not automatically updated. If *NEWSTATE* is CLOSED, the display window is closed. The display mode is initially set to CLOSED.
>
> Once the display window is open, you can update it or change its state with the mouse. Left-buttoning the display window updates it, and middle-buttoning the window brings up a menu that allows you to change the display state.
>
> > Note: DSKDISPLAY uses the value of the variable DSKDISPLAY.POSITION for the position of the lower-left corner of the disk display window when it is opened. This variable is changed if the disk display window is moved.

(SCAVENGEDSKDIRECTORY *VOLUMENAME SILENT*)   [Function]

> Rebuilds the lisp file directory for the logical volume *VOLUMENAME*. This may repair damage in the unlikely event of file system failure, signified by symptoms such as infinite looping or other strange behavior while the system is doing a directory search. Calling SCAVENGEDSKDIRECTORY will not harm an intact volume.
>
> Normally, SCAVENGEDSKDIRECTORY prints out messages as it scavenges the directory. If *SILENT* is non-NIL, these messages are not printed.
>
> > Note: Some low-level disk failures may cause "HARD DISK ERROR" errors to occur. To fix such a failure, it may be necessary to log out of Interlisp, scavenge the logical volume in question using Pilot tools, and then call SCAVENGEDSKDIRECTORY from within Interlisp. See the users guide for your computer for more information.

Floppy Disk Device

Warning: This section describes the Medley functions that control the floppy disk drive available on some computers. All of these functions may not work on all computers running Medley. For more information on using the floppy disk facilities, see the users guide for your computer.

The floppy disk drive is accessed through the device {FLOPPY}. Medley can be used to read, write, and otherwise interact with files on floppy disks through standard Interlisp input/output functions. All I/O functions such as LOAD, OPENSTREAM, READ, PRINT, GETFILEINFO, COPYFILE, etc., work with files on floppies.

Note that floppy disks are a removable storage medium. Therefore, it is only meaningful to perform I/O operations to the floppy disk drive, rather than to a given floppy disk. In this section, the phrase "the floppy" is used to mean "the floppy that is currently in the floppy disk drive."

For example, the following sequence could be used to open a file `XXX.TXT` on the floppy, print "Hello" on it, and close it:

```
(SETQ XXX (OPENSTREAM ’{FLOPPY}XXX.TXT ’OUTPUT ’NEW)
    (PRINT "Hello" XXX)
    (CLOSEF XXX)
```

(FLOPPY.MODE *MODE*)   [Function]

Medley can currently read and write files on floppies stored in a number of different formats. At any point, the floppy is considered to be in one of four "modes," which determines how it reads and writes files on the floppy. FLOPPY.MODE sets the floppy mode to the value of *MODE*, one of PILOT, HUGEPILOT, SYSOUT, or CPM, and returns the previous floppy mode. The floppy modes are interpreted as follows:

PILOT   This is the normal floppy mode, using floppies in the Xerox Pilot floppy disk format. This file format allows all of the normal Medley I/O operations. This format also supports file names with arbitrary levels of subdirectories. For example, it is possible to create a file named {FLOPPY}<Lisp>Project>FOO.TXT.

HUGEPILOT   This floppy mode is used to access files that are larger than a single floppy, stored on multiple floppies. There are some restrictions with using "huge" files. Some I/O operations are not meaningful for "huge" files. When a stream is created for output in this mode, the LENGTH file attribute must be specified when the file is opened, so that it is known how many floppies will be needed. When an output file is created, the floppy (or floppies) are automatically erased and reformatted (after confirmation from you).

HUGEPILOT mode is primarily useful for saving big files to and from floppies. For example, the following could be used to copy the file {ERIS}<Lisp>Bigfile.txt onto the huge Pilot file {FLOPPY}BigFile.save:

```
(FLOPPY.MODE ’HUGEPILOT)

(COPYFILE      ’{ERIS}<Lisp>Bigfile.txt
’{FLOPPY}BigFile.save)
```

and the following would restore the file:

```
(FLOPPY.MODE 'HUGEPILOT)
```

```
(COPYFILE        '{FLOPPY}BigFile.save
'{ERIS}<Lisp>Bigfile.txt)
```

During each copying operation, you will be prompted to insert "the next floppy" if `{ERIS}<Lisp>Bigfile.txt` takes multiple floppies.

SYSOUT   Similar to `HUGEPILOT` mode, `SYSOUT` mode is used for storing sysout files (Chapter 12) on multiple floppy disks. You are prompted to insert new floppies as they are needed.

This mode is set automatically when `SYSOUT` or `MAKESYS` is done to the floppy device: `(SYSOUT '{FLOPPY})` or `(MAKESYS '{FLOPPY})`. Notice that the file name does not need to be specifed in `SYSOUT` mode; unlike `HUGEPILOT` mode, the file name `Lisp.sysout` is always used.

Note: The procedure for loading sysout files from floppies depends on the particular computer being used. For information on loading sysout files from floppies, see the users guide for your computer.

Explicitly setting the mode to `SYSOUT` is useful when copying a sysout file to or from floppies. For example, the following can be used to copy the sysout file `{ERIS}<Lisp>Lisp.sysout` onto floppies (it is important to set the floppy mode back when done):

```
(FLOPPY.MODE 'SYSOUT)
(COPYFILE      '{ERIS}<Lisp>Lisp.sysout
'{FLOPPY})
(FLOPPY.MODE 'PILOT)
```

CPM   Medley supports the single-density single-sided (SDSS) CPM floppy format (a standard used by many computers). CPM-formatted floppies are totally different than Pilot floppies, so you should call `FLOPPY.MODE` to switch to `CPM` mode when planning to use CPM floppies. After switching to `CPM` mode, `FLOPPY.FORMAT` can be used to create CPM-formatted floppies, and the usual input/output operations work with CPM floppy files.

Note: There are a few limitations on CPM floppy format files: (1) CPM file names are limited to eight or fewer characters, with extensions of three or fewer characters; (2) CPM floppies do not have directories or version numbers; and (3) CPM files are padded out with blanks to make the file lengths multiples of 128.

(FLOPPY.FORMAT *NAME AUTOCONFIRMFLG SLOWFLG*)   [Function]

FLOPPY.FORMAT erases and initializes the track information on a floppy disk. This must be done when new floppy disks are to be used for the first time. This can also be used to erase the information on used floppy disks.

NAME should be a string that is used as the name of the floppy (106 characters max). This name can be read and set using FLOPPY.NAME (below).

If *AUTOCONFIRMFLG* is NIL, you will be prompted to confirm erasing the floppy, if it appears to contain valid information. If *AUTOCONFIRMFLG* is T, you are not prompted to confirm.

If *SLOWFLG* is NIL, only the Pilot records needed to give your floppy an empty directory are written. If *SLOWFLG* is T, FLOPPY.FORMAT will completely erase the floppy, writing track information and critical Pilot records on it. *SLOWFLG* should be set to T when formatting a brand-new floppy.

Note: Formatting a floppy is a very compute-intensive operation for the I/O hardware. Therefore, the cursor may stop tracking the mouse and keystrokes may be lost while formatting a floppy. This behavior goes away when the formatting is finished.

Warning: The floppy mode set by FLOPPY.MODE (above) affects how FLOPPY.FORMAT formats the floppy. If the floppy is going to be used in Pilot mode, it should be formatted under (FLOPPY.MODE 'PILOT). If it is to be used as a CMP floppy, it should be formatted under (FLOPPY.MODE 'CPM). The two types of formatting are incompatible.

(FLOPPY.NAME *NAME*)   [Function]

If *NAME* is NIL, returns the name stored on the floppy disk. If *NAME* is non-NIL, then the name of the floppy disk is set to *NAME*.

(FLOPPY.FREE.PAGES)   [Function]

Returns the number of unallocated free pages on the floppy disk in the floppy disk drive.

Note: Pilot floppy files are represented by contiguous pages on a floppy disk. If you are creating and deleting a lot of files on a floppy, it is advisable to keep such a floppy less than 75 percent full.

(FLOPPY.CAN.READP)   [Function]

Returns non-NIL if there is a floppy in the floppy drive.

Note: `FLOPPY.CAN.READP` does not provide any debouncing (protection against not fully closing the floppy drive door). It may be more useful to use `FLOPPY.WAIT.FOR.FLOPPY` (below).

(`FLOPPY.CAN.WRITEP`)   [Function]

Returns non-NIL if there is a floppy in the floppy drive and the floppy drive can write on this floppy.

It is not possible to write on a floppy disk if the "write-protect notch" on the floppy disk is punched out.

(`FLOPPY.WAIT.FOR.FLOPPY` *NEWFLG*)   [Function]

If *NEWFLG* is NIL, waits until a floppy is in the floppy drive before returning.

If *NEWFLG* is T, waits until the existing floppy in the floppy drive, if any, is removed, then waits for a floppy to be inserted into the drive before returning.

(`FLOPPY.SCAVENGE`)   [Function]

Attempts to repair a floppy whose critical records have become confused (causing errors when file operations are attempted). May also retrieve accidently-deleted files, provided they haven't been overwritten by new files.

(`FLOPPY.TO.FILE` *TOFILE*)   [Function]

Copies the entire contents of the floppy to the "floppy image" file *TOFILE*, which can be on a file server, local disk, etc. This can be used to create a centralized copy of a floppy, that different users can copy to their own floppy disks (using `FLOPPY.FROM.FILE`).

Note: A floppy image file for an 8-inch floppy is about 2500 pages long, regardless of the number of pages in use on the floppy.

(`FLOPPY.FROM.FILE` *FROMFILE*)   [Function]

Copies the "floppy image" file *FROMFILE* to the floppy. *FROMFILE* must be a file produced by `FLOPPY.TO.FILE`.

(`FLOPPY.ARCHIVE` *FILES NAME*)   [Function]

`FLOPPY.ARCHIVE` formats a floppy inserted into the floppy drive, giving the floppy the name *NAME*#1. `FLOPPY.ARCHIVE` then copies each file in *FILES* to the freshly formatted floppy. If the first floppy fills up, `FLOPPY.ARCHIVE` uses multiple floppies (named *NAME*#2, *NAME*#3, etc.), each time prompting you to insert a new floppy.

The function `DIRECTORY` (see below) is convenient for generating a list of files to archive. For example,

```
(FLOPPY.ARCHIVE
    (DIRECTORY '{ERIS}<Lisp>Project>*)
    'Project)
```

will archive all files on the directory `{ERIS}<Lisp>Project>` to floppies (named `Project#1`, `Project#2`, etc.).

(`FLOPPY.UNARCHIVE` *HOST/DIRECTORY*)   [Function]

`FLOPPY.UNARCHIVE` copies all files on the current floppy to the directory *HOST/DIRECTORY*. For example, (`FLOPPY.UNARCHIVE` '`{ERIS}<Lisp>Project>`) will copy each file on the current floppy to the directory `{ERIS}<Lisp>Project>`. If there is more than one floppy to restore from archive, `FLOPPY.UNARCHIVE` should be called on each floppy disk.

I/O Operations to and from Strings

It is possible to treat a string as if it were the contents of a file by using the following function:

(`OPENSTRINGSTREAM` *STR ACCESS*)   [Function]

Returns a stream that can be used to access the characters of the string *STR*. *ACCESS* may be either `INPUT`, `OUTPUT`, or `BOTH`; `NIL` defaults to `INPUT`. The stream returned may be used exactly like a file opened with the same access, except that output operations may not extend past the end of the original string. Also, string streams do not appear in the value of (`OPENP`).

For example, after performing

```
(SETQ STRM (OPENSTRINGSTREAM "THIS 2 (IS A LIST)"))
```

the following succession of reads could occur:

```
(READ STRM)  => THIS
   (RATOM STRM) => 2
   (READ STRM)  => (IS A LIST)
   (EOFP STRM)  => T
```

Compatibility Note: In Interlisp-10 it was possible to take input from a string simply by passing the string as the *FILE* argument to an input function. In order to maintain compatibility with this feature, Medley provides the same capability. This not terribly clean feature persists in the present implementation to give users time to convert old code. This means that strings are *not* equivalent to symbols when specifying a file name as a stream argument. In a future release, the old Interlisp-10 string-reading feature will be decommissioned, and `OPENSTRINGSTREAM` will be the only way to perform I/O on a string.

Temporary Files and the CORE Device

Many operating systems have a notion of "scratch file", a file typically used as temporary storage for data most naturally maintained in the form of a file, rather than some other data structure. A scratch

file can be used as a normal file in most respects, but is automatically deleted from the file system after its useful life is up, e.g., when the job terminates, or you log out. In normal operation, you need never explicitly delete such files, since they are guaranteed to disappear soon.

A similar functionality is provided in Medley by core-resident files. Core-resident files are on the device `CORE`. The directory structure for this device and all files on it are represented completely within your virtual memory. These files are treated as ordinary files by all file operations; their only distinguishing feature is that all trace of them disappears when the virtual memory is abandoned.

Core files are opened and closed by name the same as any other file, e.g., `(OPENSTREAM '{CORE}<FOO>FIE.DCOM 'OUTPUT)`. Directory names are completely optional, so files can also have names of the form `{CORE}NAME.EXT`. Core files can be enumerated by `DIRECTORY` (see below). While open, they are registered in `(OPENP)`. They do consume virtual memory space, which is only reclaimed when the file is deleted. Some caution should thus be used when creating large `CORE` files. Since the virtual memory of an Medley workstation usually persists far longer than the typical process on a mainframe computer, it is still important to delete `CORE` files after they are no longer in use.

For many applications, the name of the scratch file is irrelevant, and there is no need for anyone to have access to the file independent of the program that created it. For such applications, `NODIRCORE` files are preferable. Files created on the device lisp `NODIRCORE` are core-resident files that have no name and are registered in no directory. These files "disappear", and the resources they consume are reclaimed, when all pointers to the file are dropped. Hence, such files need never be explicitly deleted or, for that matter, closed. The "name" of such a file is simply the stream object returned from `(OPENSTREAM '{NODIRCORE} 'OUTPUT)`, and it is this stream object that must be passed to all input/output operations, including `CLOSEF` and any calls to `OPENSTREAM` to reopen the file.

> `(COREDEVICE NAME NODIRFLG)` [Function]
>
> > Creates a new device for core-resident files and assigns *NAME* as its device name. Thus, after performing `(COREDEVICE 'FOO)`, one can execute `(OPENSTREAM '{FOO}BAR 'OUTPUT)` to open a file on that device. Medley is initialized with the single core-resident device named `CORE`, but `COREDEVICE` may be used to create any number of logically distinct core devices.
> >
> > If *NODIRFLG* is non-`NIL`, a core device that acts like `{NODIRCORE}` is created.
> >
> > Compatibility note: In Interlisp-10, it was possible to create scratch files by using file names with suffixes `;S` or `;T`. In Medley, these suffixes in file names are simply ignored when output is directed to a particular host or device. However, the function `PACKFILENAME.STRING` is defined to default the device name to `CORE` if the file has the `TEMPORARY` attribute and no explicit host is provided.

NULL Device

The NULL device provides a source of content-free "files". (OPENSTREAM '{NULL} 'OUTPUT) creates a stream that discards all output directed at it. (OPENSTREAM '{NULL} 'INPUT) creates a stream that is perpetually at end-of-file (i.e., has no input).

Deleting, Copying, and Renaming Files

(DELFILE *FILE*)   [Function]

> Deletes *FILE* if possible. The file must be closed. Returns the full name of the file if deleted, else NIL. Recognition mode for *FILE* is OLDEST, i.e., if *FILE* does not have a version number specified, then DELFILE deletes the oldest version of the file.

(COPYFILE *FROMFILE TOFILE*)   [Function]

> Copies *FROMFILE* to a new file named *TOFILE*. The source and destination may be on any combination of hosts/devices. COPYFILE attempts to preserve the TYPE and CREATIONDATE where possible. If the original file's file type is unknown, COPYFILE attempts to infer the type (file type is BINARY if any of its 8-bit bytes have their high bit on).

> COPYFILE uses COPYCHARS (Chapter 25) if the source and destination hosts have different EOL conventions. Thus, it is possible for the source and destination files to be of different lengths.

(RENAMEFILE *OLDFILE NEWFILE*)   [Function]

> Renames *OLDFILE* to be *NEWFILE*. Causes an error, FILE NOT FOUND if *FILE* does not exist. Returns the full name of the new file, if successful, else NIL if the rename cannot be performed.

> If *OLDFILE* and *NEWFILE* are on the same host/device, and the device implements a renaming primitive, RENAMEFILE can be very fast. However, if the device does not know how to rename files in place, or if *OLDFILE* and *NEWFILE* are on different devices, RENAMEFILE works by copying *OLDFILE* to *NEWFILE* and then deleting *OLDFILE*.

Searching File Directories

DIRECTORIES   [Variable]

> Global variable containing the list of directories searched (in order) by SPELLFILE and FINDFILE (below) when not given an explicit *DIRLST* argument. In this list, the atom NIL stands for the login directory (the value of LOGINHOST/DIR), and the atom T stands for the currently connected directory. Other elements should be *full* directory specifications, e.g., {TWENTY}PS:<LISPUSERS>, not merely LISPUSERS.

LISPUSERSDIRECTORIES   [Variable]

Global variable containing a list of directories to search for "library" package files. Used by the FILES file package command (Chapter 17).

(SPELLFILE *FILE NOPRINTFLG NSFLG DIRLST*)   [Function]

Searches for the file name *FILE*, possibly performing spelling correction (see Chapter 20). Returns the corrected file name, if any, otherwise NIL.

If *FILE* has a directory field, SPELLFILE attempts spelling correction against the files in that particular directory. Otherwise, SPELLFILE searches for the file on the directory list *DIRLST* before attempting any spelling correction.

If *NOPRINTFLG* is NIL, SPELLFILE asks you to confirm any spelling correction done, and prints out any files found, even if spelling correction is not done. If *NOPRINTFLG* = T, SPELLFILE does not do any printing, nor ask for approval.

If *NSFLG* = T (or NOSPELLFLG = T, see Chapter 20), no spelling correction is attempted, though searching through *DIRLST* still occurs.

DIRLST is the list of directories searched if *FILE* does not have a directory field. If *DIRLST* is NIL, the value of the variable DIRECTORIES is used.

> Note: If *DIRLST* is NIL, and *FILE* is not found by searching the directories on DIRECTORIES, but the root name of *FILE* has a FILEDATES property (Chapter 17) indicating that a file by that name has been loaded, then the directory indicated in the FILEDATES property is searched, too. This additional search is not done if *DIRLST* is non-NIL.

ERRORTYPELST (Chapter 14) initially contains the entry ((23 (SPELLFILE (CADR ERRORMESS) NIL NOFILESPELLFLG))), which causes SPELLFILE to be called in case of a FILE NOT FOUND error. If the variable NOFILESPELLFLG is T (its initial value), then spelling correction is not done on the file name, but DIRECTORIES is still searched. If SPELLFILE is successful, the operation will be reexecuted with the new (corrected) file name.

(FINDFILE *FILE NSFLG DIRLST*)   [Function]

Uses SPELLFILE to search for a file named *FILE*. If it finds one, returns its full name, with no user interaction. Specifically, it calls (SPELLFILE *FILE* T *NSFLG DIRLST*), after first performing two simple checks: If *FILE* has an explicit directory, it checks to see if a file so named exists, and if so returns that file. If *DIRLST* is NIL, it looks for *FILE* on the connected directory before calling SPELLFILE.

Listing File Directories

The function `DIRECTORY` allows you to conveniently specify and/or program a variety of directory operations:

> (DIRECTORY *FILES COMMANDS DEFAULTEXT DEFAULTVERS*) [Function]
>
>> Returns, lists, or performs arbitrary operations on all files specified by the "file group" *FILES*. A file group has the form of a regular file name, except that the character * can be used to match any number of characters, including zero, in the file name. For example, the file group A*B matches all file names beginning with the character A and ending with the character B. The file group *.DCOM matches all files with an extension of DCOM.
>>
>> If *FILES* does not contain an explicit extension, it is defaulted to *DEFAULTEXT*; if *FILES* does not contain an explicit version, it is defaulted to *DEFAULTVERS*. *DEFAULTEXT* and *DEFAULTVERS* themselves default to *. If the period or semicolon preceding the omitted extension or version, respectively, is present, the field is explicitly empty and no default is used. All other unspecified fields default to *. Null version is interpreted as "highest". Thus *FILES* = * or *.* or *.*;* enumerates all files on the connected directory; *FILES* = *. or *.;* enumerates all versions of files with null extension; *FILES* = *.; enumerates the highest version of files with null extension; and *FILES* = *.*; enumerates the highest version of all files. If *FILES* is NIL, it defaults to *.*;*.
>>
>>> Note: Some hosts/devices are not capable of supporting "highest version" in enumeration. Such hosts instead enumerate *all* versions.
>>
>> For each file that matches the file group *FILES*, the "file commands" in *COMMANDS* are executed in order. Some of the file commands allow aborting the command processing for a given file, effectively filtering the list of files. The interpretation of the different file commands is described below. If *COMMANDS* is NIL, it defaults to (COLLECT), which collects the matching file names in a list and returns it as the value of DIRECTORY.

The "file commands" in *COMMANDS* are interpreted as follows:

P	Prints the file's name. For readability, DIRECTORY strips the directory from the name, printing it once as a header in front of each set of consecutive files on the same directory.
PP	Prints the file's name without a version number.
a string	Prints the string.
READDATE, WRITEDATE CREATIONDATE, SIZE LENGTH, BYTESIZE PROTECTION, AUTHOR TYPE	Prints the appropriate information returned by GETFILEINFO (see above).

COLLECT Adds the full name of this file to an accumulating list, which will be returned as the value of DIRECTORY.

COUNTSIZE Adds the size of this file to an accumulating sum, which will be returned as the value of DIRECTORY.

DELETE Deletes the file.

DELVER If this file is not the highest version of files by its name, delete it.

PAUSE Waits until you type any character before proceeding with the rest of the commands (good for display if you want to ponder).

The following commands are predicates to filter the list. If the predicate is not satisfied, then processing for this file is aborted and no further commands (such as those above) are executed for this file.

Note: if the P and PP commands appear in *COMMANDS* ahead of any of the filtering commands below except PROMPT, they are postponed until after the filters. Thus, assuming the caller has placed the attribute options after the filters as well, no printing occurs for a file that is filtered out. This is principally so that functions like DIR (below) can both request printing and pass arbitrary commands through to DIRECTORY, and have the printing happen in the appropriate place.

PROMPT *MESS* Prompts with the yes/no question *MESS*; if user responds with No, abort command processing for this file.

OLDERTHAN *N* Continue command processing if the file hasn't been referenced (read or written) in *N* days. *N* can also be a string naming an explicit date and time since which the file must not have been referenced.

NEWERTHAN *N* Continue command processing if the file has been written within the last *N* days. *N* can also be a string naming an explicit date and time. Note that this is not quite the complement of OLDERTHAN, since it ignores the read date.

BY *USER* Continue command processing if the file was last written by the given user, i.e., its AUTHOR attribute matches (case insensitively) *USER*.

@ *X* *X* is either a function of one argument (*FILENAME*), or an arbitrary expression which uses the variable FILENAME freely. If *X* returns NIL, abort command processing for this file.

The following two commands apply not to any particular file, but globally to the manner in which directory information is printed.

OUT *FILE*   Directs output to *FILE*.

COLUMNS *N*   Attempts to format output in *N* columns (rather than just 1).

DIRECTORY uses the variable DIRCOMMANDS as a spelling list to correct spelling and define abbreviations and synonyms (see Chapter 20). Currently the following abbreviations are recognized:

AU	=>	AUTHOR
–	=>	PAUSE
COLLECT?	=>	PROMPT " ? " COLLECT
DA		
DATE	=>	CREATIONDATE
TI	=>	WRITEDATE
DEL	=>	DELETE
DEL?		
DELETE?	=>	PROMPT " delete? " DELETE
OLD	=>	OLDERTHAN 90
PR	=>	PROTECTION
SI	=>	SIZE
VERBOSE	=>	AUTHOR CREATIONDATE SIZE READDATE WRITEDATE

(FILDIR *FILEGROUP*)   [Function]

Obsolete synonym of (DIRECTORY *FILEGROUP*).

(DIR *FILEGROUP* COM$_1$ ... COM$_N$)   [NLambda NoSpread Function]

Convenient form of DIRECTORY for use in type-in at the executive. Performs (DIRECTORY '*FILEGROUP* '(P COM$_1$ ... COM$_N$)).

(NDIR *FILEGROUP* COM$_1$ ... COM$_N$)   [NLambda NoSpread Function]

Version of DIR that lists the file names in a multi-column format. Also, by default only lists the most recent version of files (unless *FILEGROUP* contains an explicit version).

File Servers

A file server is a shared resource on a local communications network which provides large amounts of file storage. Different file servers honor a variety of access protocols. Medley supports the following

protocols: PUP-FTP, PUP-Leaf, and NS Filing. In addition, there are library packages available that support other communications protocols, such as TCP/IP and RS232.

With the exception of the RS232-based protocols, which exist only for file transfer, these network protocols are integrated into the Medley file system to allow files on a file server to be treated in much the same way files are accessed on local devices, such as the disk. Thus, it is possible to call `OPENSTREAM` on the file `{ERIS}<LISP>FOO.DCOM;3` and read from it or write to it just as if the file had been on the local disk (`{DSK}<LISP>FOO.DCOM;3`), rather than on a remote server named ERIS. However, the protocols vary in how much control they give the workstation over file system operations. Hence, some restrictions apply, as described in the following sections.

PUP File Server Protocols

There are two file server protocols in the family of PUP protocols: Leaf and FTP. Some servers support both, while others support only one of them. Medley uses whichever protocol is more appropriate for the requested operation.

Leaf is a random access protocol, so files opened using these protocols are `RANDACCESSP`, and thus most normal I/O operations can be performed. However, Leaf does not support directory enumeration. Hence, `DIRECTORY` cannot be used on a Leaf file server unless the server also supports FTP. In addition, Leaf does not supply easy access to a file's attributes. `INFILEP` and `GETFILEINFO` have to open the file for input in order to obtain their information, and hence the file's read date will change, even though the semantics of these functions do not imply it.

FTP is a file transfer protocol that only permits sequential access to files. However, most implementations of it are considerably more efficient than Leaf. Medley uses FTP in preference to Leaf whenever the call to `OPENSTREAM` requests sequential access only. In particular, the functions `SYSOUT` and `COPYFILE` open their files for sequential access. If a file server supports FTP but for some reason it is undesirable for Lisp to use it, one can set the internal variable `\FTPAVAILABLE` to `NIL`.

The system normally maintains a Leaf connection to a host in the background. This connection can be broken by calling (`BREAKCONNECTION HOST`). Any subsequent reference to files on that host will re-establish the connection. The principal use for this function arises when you interrupt a file operation in such a way that the file server thinks the file is open but Lisp thinks it is closed (or not yet open). As a result, the next time Lisp tries to open the file, it gets a file busy error.

Xerox NS File Server Protocols

Interlisp supports file access to Xerox 803x file servers, using the Filing Protocol built on Xerox Network Systems protocols. Medley determines that a host is an NS File Server by the presense of a colon in its name, e.g., `{PHYLEX:}`. The general format of NS fileserver device names is `{SERVERNAME:DOMAIN:ORGANIZATION}`; the device specification for an 8000-series product in general includes the ClearingHouse domain and organization. If domain and organization are not supplied directly, then they are obtained from the defaults, which themselves are found by consulting

the nearest ClearingHouse if you have not defined them in an init file. However, note that the server name must still have a colon in it to distinguish it from other types of host names (e.g., PUP server names).

NS file servers in general permit arbitrary characters in file names. You should be cognizant of file name quoting conventions, and the fact that any file name presented as a symbol needs to have characters of significance to the reader, such as space, escaped with a %. Of course, one can always present the file name as a string, in which case only the quoting conventions are important.

NS file servers support a true hierarchical file system, where subdirectories are just another kind of file, which needs to be explicitly created. In Interlisp, subdirectories are created automatically as needed: A call to `OPENFILE` to create a file in a non-existent subdirectory automatically creates the subdirectory. `CONN` to a non-existent subdirectory asks you whether to create the directory. For those using Star software, a directory corresponds to a "File Drawer," while a subdirectory corresponds to a "File Folder."

Because of their hierarchical structure, NS directories can be enumeerated to arbitrary levels. The default is to enumerate all the files (the leaves of the tree), omitting the subdirectory nodes themselves. This default can be changed by the following variable:

> `FILING.ENUMERATION.DEPTH`   [Variable]

> > This variable is either a number, specifying the number of levels deep to enumeerate, or `T`, meaning enumerate to all levels. In the former case, when the enumeration reaches the specified depth, only the subdirectory name rooted at that level is listed, and none of its descendants is listed. When `FILING.ENUMERATION.DEPTH` is `T`, all files are listed, and no subdirectory names are listed. `FILING.ENUMERATION.DEPTH` is initially `T`.

> > Independent of `FILING.ENUMERATION.DEPTH`, a request to enumerate the top-level of a file server's hierarchy lists only the top level, i.e., assumes a depth of 1. For example, `(DIRECTORY '{PHYLEX:})` lists exactly the top-level directories of the server `PHYLEX:`.

NS file servers do not currently support random access. Therefore, `SETFILEPTR` of an NS file generally causes an error. However, `GETFILEPTR` returns the correct character position for open files on NS file servers. In addition, `SETFILEPTR` works in the special case where the file is open for input, and the file pointer is being set forward. In this case, the intervening charactgers are automatically read.

Even while Interlisp has no file open on an NS Server, the system maintains a "session" with the server for a while in order to improve the speed of subsequent requests to the server. While this session is open, it is possible for some nodes of the server's file system to appear "busy" or inaccessible to certain clients on other workstations (such as Star). If this happens, the following function can be used to terminate any open sessions immediately.

> `(BREAK.NSFILING.CONNECTION `*HOST*`)`   [Function]

> > Closes any open connections to NS file server *HOST*.

Operating System Designations

Some of the network server protocols are implemented on more than one kind of foreign host. Such hosts vary in their conventions for logging in, naming files, representing end-of-line, etc. In order for Interlisp to communicate gracefully with all these hosts, it is necessary that the variable NETWORKOSTYPES be set correctly.The following functions are now considered obsolete, but are provided for backwards compatibility:

NETWORKOSTYPES [Variable]

An association-list that associates a host name with its operating system type. Elements in this list are of the form (*HOSTNAME . TYPE*). For example, (MAXC2 . TENEX). The operating system types currently known to Lisp are TENEX, TOPS20, UNIX, and VMS. The host names in this list should be the "canonical" host name, represented as an uppercase atom. For PUP and NS hosts, the function CANONICAL.HOSTNAME (below) can be used to determine which of several aliases of a server is the canonical name.

(CANONICAL.HOSTNAME *HOSTNAME*) [Function]

Returns the "canonical" name of the server *HOSTNAME*, or NIL if *HOSTNAME* is not the name of a server.

Logging In

Most file servers require a user name and password for access. Medley maintains an ephemeral database of user names and passwords for each host accessed recently. The database vanishes when LOGOUT, SAVEVM, SYSOUT, or MAKESYS is executed, so that the passwords remain secure from any subsequent user of the same virtual memory image. Medley also maintains a notion of the "default" user name and password, which are generally those with which you initially log in.

When a file server for which the system does not yet have an entry in its password database requests a name and password, the system first tries the default user name and password. If the file server does not recognize that name/password, the system prompts you for a name and password to use for that host. It suggests a default name:

```
{ERIS} Login: Green
```

which you can accept by pressing [Return}, or replace the name by typing a new name or backspacing over it. Following the name, you are prompted for a password:

```
{ERIS} Login: Verdi (password)
```

which is not echoed, terminated by another [Return]. This information is stored in the password database so that you are prompted only once, until the database is again cleared.

Medley also prompts for password information when a protection violation occurs on accessing a directory on certain kinds of servers that support password-protected directories. Some such servers allow one to protect a file in a way that is inaccessible to even its owner until the file's protection is changed. In such cases, no password would help, and the system causes the normal PROTECTION VIOLATION error.

You can abort a password interaction by typing the ERROR interrupt, initially Cosntrol-E. This generally either causes a PROTECTION VIOLATION error, if the password was requested in order to gain access to a protected file on an otherwise accessible server; or to act as though the server did not exist, in the case where the password was needed to gain any access to the server.

(LOGIN *HOSTNAME FLG DIRECTORY MSG*) [Function]

Forces Medley to ask for your name and password to be used when accessing host *HOSTNAME*. Any previous login information for *HOSTNAME* is overridden. If *HOSTNAME* is NIL, it overrides login information for all hosts and resets the default user name and password to be those typed in by you. The special value *HOSTNAME* = NS:: is used to obtain the default user name and password for all logins for NS Servers.

If *FLG* is the atom QUIET, only prompts you if there is no cached information for *HOSTNAME*.

If *DIRECTORY* is specified, it is the name of a directory on *HOSTNAME*. In this case, the information requested is the "connect" password for that directory. Connect passwords for any number of different directories on a host can be maintained.

If *MSG* is non-NIL, it is a message (a string) to be printed before the name and password information is requested.

LOGIN returns the user name with which you completed the login.

(SETPASSWORD *HOST USER PASSWORD DIRECTORY*) [Function]

Sets the values in the internal password database exactly as if the strings *USER* and *PASSWORD* were typed in via (LOGIN *HOST* NIL *DIRECTORY*).

(SETUSERNAME *NAME*) [Function]

Sets the default uer name to *NAME*.

(USERNAME *FLG STRPTR PRESERVECASE*) [Function]

If *FLG* = NIL, returns the default user name. This is the only value of *FLG* that is meaningful in Medley.

USERNAME returns the value as a string, unless *STRPTR* is T, in which case USERNAME returns the value as an atom. The name is returned in uppercase, unless *PRESERVECASE* is true.

Abnormal Conditions

If Medley tries to access a file and does not get a response from the file server in a areasonable period of time, it prints a message that the file server is not responding, and keeps trying. If the file server has actually crashed, this may continue indefinitely. A Control-E or similar interrupt aborts out of this state.

If the file server crashes but is restarted before you attempt to do anything, file operations will usually proceed normally, except for a brief pause while Medley tries to re-establish any connections it had open before the crash. However, this is not always possible. For example, when a file is open for sequential output and the server crashes, there is no way to recover the output already written, since it vanished with the crash. In such cases, the system will cause an error such as `Connection Lost.`

`LOGOUT` closes any file server connections that are currently open. On return, it attempts to re-establish connections for any files that were open before logging out. If a file has disappeared or been modified, Medley reports this fact. Files that were open for sequential access generally cannot be reopened after `LOGOUT`.

Interlisp supports simultaneous access to the same server from different processes and permits overlapping of Lisp computation with file server operations, allowing for improved performance. However, as a corollary of this, a file is not closed the instant that `CLOSEF` returns; Interlisp closes the file "in the bckground". It is therefore very important that you exit Interlisp via `(LOGOUT)` or `(LOGOUT T)`, rather than boot the machine.

On rare occasions, the Ethernet may appear completely unresponsive, due to Interlisp having gotten into a bad state. Type `(RESTART.ETHER)` to reinitialize Lisp's Ethernet driver(s), just as when the Lisp system is started up following a `LOGOUT`, `SYSOUT`, etc.

24. INPUT/OUTPUT FUNCTIONS

This chapter describes the standard I/O functions used for reading and printing characters and Interlisp expressions on files and other streams. First, the primitive input functions are presented, then the output functions, then functions for random-access operations (such as searching a file for a given stream, or changing the "next-character" pointer to a position in a file). Next, the PRINTOUT statement is documented (see below), which provides an easy way to write complex output operations. Finally, read tables, used to parse characters as Interlisp expressions, are documented.

Specifying Streams for Input/Output Functions

Most of the input/output functions in Interlisp-D have an argument named *STREAM* or *FILE*, specifying on which open stream the function's action should occur (the name *FILE* is used in older functions that predate the concept of stream; the two should, however, be treated synonymously). The value of this argument should be one of the following:

a stream
: An object of type STREAM, as returned by OPENSTREAM (Chapter 23) or other stream-producing functions, is always the most precise and efficient way to designate a stream argument.

T
: The litatom T designates the terminal input or output stream of the currently running process, controlling input from the keyboard and output to the display screen. For functions where the direction (input or output) is ambiguous, T is taken to designate the terminal output stream. The T streams are always open; they cannot be closed.

: The terminal output stream can be set to a given window or display stream by using TTYDISPLAYSTREAM (Chapter 28). The terminal input stream cannot be changed. For more information on terminal I/O, see Chapter 30.

NIL
: The litatom NIL designates the "primary" input or output stream. These streams are initially the same as the terminal input/output streams, but they can be changed by using the functions INPUT and OUTPUT.

: For functions where the direction (input or output) is ambiguous, e.g., GETFILEPTR, the argument NIL is taken to mean the primary input stream, if that stream is not identical to the terminal input stream, else the primary output stream.

a window
: Uses the display stream of the window . Valid for output only.

a file name
: As of this writing, the name of an open file (as a litatom) can be used as a stream argument. However, there are inefficiencies and possible future incompatibilities associated with doing so. See Chapter 24 for details.

(**GETSTREAM** *FILE ACCESS*)   [Function]

> Coerces the argument *FILE* to a stream by the above rules. If *ACCESS* is INPUT, OUTPUT, or BOTH, produces the stream designated by *FILE* that is open for *ACCESS*. If *ACCESS*=NIL, returns a stream for *FILE* open for any kind of input/output (see the list above for the ambiguous cases). If *FILE* does not designate a stream open in the specified mode, causes an error, FILE NOT OPEN.

(**STREAMP** *X*)   [Function]

> Returns *X* if *X* is a STREAM, otherwise NIL.

Input Functions

> While the functions described below can take input from any stream, some special actions occur when the input is from the terminal (the T input stream, see above). When reading from the terminal, the input is buffered a line at a time, unless buffering has been inhibited by CONTROL (Chapter 30) or the input is being read by READC or PEEKC. Using specified editing characters, you can erase a character at a time, a word at a time, or the whole line. The keys that perform these editing functions are assignable via SETSYNTAX, with the initial settings chosen to be those most natural for the given operating system. In Interlisp-D, the initial settings are as follows: characters are deleted one at a time by Backspace; words are erased by control-W; the whole line is erased by Control-Q.

> On the Interlisp-D display, deleting a character or a line causes the characters to be physically erased from the screen. In Interlisp-10, the deleting action can be modified for various types of display terminals by using DELETECONTROL (Chapter 30).

> Unless otherwise indicated, when the end of file is encountered while reading from a file, all input functions generate an error, END OF FILE. Note that this does not close the input file. The ENDOFSTREAMOP stream attribute (Chapter 24) is useful for changing the behavior at end of file.

> Most input functions have a *RDTBL* argument, which specifies the read table to be used for input. Unless otherwise specified, if *RDTBL* is NIL, the primary read table is used.

> If the *FILE* or *STREAM* argument to an input function is NIL, the primary input stream is used.

(**INPUT** *FILE*)   [Function]

> Sets *FILE* as the primary input stream; returns the old primary input stream. *FILE* must be open for input.

> (INPUT) returns the current primary input stream, which is not changed.

> Note: If the primary input stream is set to a file, the file's full name, rather than the stream itself, is returned. See discussion in Chapter 24.

(**READ** *FILE RDTBL FLG*) [Function]

> Reads one expression from *FILE*. Atoms are delimited by the break and separator characters as defined in *RDTBL*. To include a break or separator character in an atom, the character must be preceded by the character `%`, e.g., `AB%(C` is the atom `AB(C`, `%%` is the atom `%`, `%`*control-K* is the atom Control-K. For input from the terminal, an atom containing an interrupt character can be input by typing instead the corresponding alphabetic character preceded by Control-V, e.g., `^VD` for Control-D.

> Strings are delimited by double quotes. To input a string containing a double quote or a `%`, precede it by `%`, e.g., `"AB%"C"` is the string `AB"C`. Note that `%` can always be typed even if next character is not "special", e.g., `%A%B%C` is read as `ABC`.

> If an atom is interpretable as a number, `READ` creates a number, e.g., `1E3` reads as a floating point number, `1D3` as a literal atom, `1.0` as a number, `1,0` as a literal atom, etc. An integer can be input in a non-decimal radix by using syntax such as `123Q`, `|b10101`, `|5r1234` (see Chapter 7). The function `RADIX`, sets the radix used to print integers.

> When reading from the terminal, all input is line-buffered to enable the action of the backspacing control characters, unless inhibited by `CONTROL` (Chapter 30). Thus no characters are actually seen by the program until a carriage-return (actually the character with terminal syntax class `EOL`, see Chapter 30), is typed. However, for reading by `READ`, when a matching right parenthesis is encountered, the effect is the same as though a carriage-return were typed, i.e., the characters are transmitted. To indicate this, Interlisp also prints a carriage-return line-feed on the terminal. The line buffer is also transmitted to `READ` whenever an `IMMEDIATE` read macro character is typed (see below).

> *FLG*=T suppresses the carriage-return normally typed by `READ` following a matching right parenthesis. (However, the characters are still given to `READ`; i.e., you do not have to type the carriage-return.)

(**RATOM** *FILE RDTBL*) [Function]

> Reads in one atom from *FILE*. Separation of atoms is defined by *RDTBL*. `%` is also defined for `RATOM`, and the remarks concerning line-buffering and editing control characters also apply.

> If the characters comprising the atom would normally be interpreted as a number by `READ`, that number is returned by `RATOM`. Note however that `RATOM` takes no special action for `"` whether or not it is a break character, i.e., `RATOM` never makes a string.

(**RSTRING** *FILE RDTBL*) [Function]

> Reads characters from *FILE* up to, but not including, the next break or separator character, and returns them as a string. Backspace, Control-W, Control-Q, Control-V, and `%` have the same effect as with `READ`.

Note that the break or separator character that terminates a call to `RATOM` or `RSTRING` is *not* read by that call, but remains in the buffer to become the first character seen by the next reading function that

is called. If that function is RSTRING, it will return the null string. This is a common source of program bugs.

(**RATOMS** *A FILE RDTBL*)   [Function]

    Calls RATOM repeatedly until the atom *A* is read. Returns a list of the atoms read, not including *A*.

(**RATEST** *FLG*)   [Function]

    If *FLG* = T, RATEST returns T if a separator was encountered immediately prior to the atom returned by the last RATOM or READ, NIL otherwise.

    If *FLG* = NIL, RATEST returns T if last atom read by RATOM or READ was a break character, NIL otherwise.

    If *FLG* = 1, RATEST returns T if last atom read (by READ or RATOM) contained a % used to quote the next character (as in %[or %A%B%C), NIL otherwise.

(**READC** *FILE RDTBL*)   [Function]

    Reads and returns the next character, including %, ", etc, i.e., is not affected by break or separator characters. The action of READC is subject to line-buffering, i.e., READC does not return a value until the line has been terminated even if a character has been typed. Thus, the editing control characters have their usual effect. *RDTBL* does not directly affect the value returned, but is used as usual in line-buffering, e.g., determining when input has been terminated. If (CONTROL T) has been executed (Chapter 30), defeating line-buffering, the *RDTBL* argument is irrelevant, and READC returns a value as soon as a character is typed (even if the character typed is one of the editing characters, which ordinarily would never be seen in the input buffer).

(**PEEKC** *FILE*)   [Function]

    Returns the next character, but does not actually read it and remove it from the buffer. If reading from the terminal, the character is echoed as soon as PEEKC reads it, even though it is then "put back" into the system buffer, where Backspace, Control-W, etc. could change it. Thus it is possible for the value returned by PEEKC to "disagree" in the first character with a subsequent READ.

(**LASTC** *FILE*)   [Function]

    Returns the last character read from *FILE*. LASTC can return an incorrect result when called immediatley following a PEEKC on a file that contains run-coded NS characters.

(**READCCODE** *FILE RDTBL*)   [Function]

    Returns the next character *code* from *STREAM*; thus, this operation is equivalent to, but more efficient than, (CHCON1 (READC *FILE RDTBL*)).

(**PEEKCCODE** *FILE*)   [Function]

> Returns, without consuming, the next character *code* from *STREAM*; thus, this operation is equivalent to, but more efficient than, (CHCON1 (PEEKC *FILE*)).

(**BIN** *STREAM*)   [Function]

> Returns the next byte from *STREAM*. This operation is useful for reading streams of binary, rather than character, data.

> Note: BIN is similar to READCCODE, except that BIN always reads a single byte, whereas READCCODE reads a "character" that can consist of more than one byte, depending on the character and its encoding.

READ, RATOM, RATOMS, PEEKC, READC all wait for input if there is none. The only way to test whether or not there is input is to use READP:

(**READP** *FILE FLG*)   [Function]

> Returns T if there is anything in the input buffer of *FILE*, NIL otherwise. This operation is only interesting for streams whose source of data is dynamic, e.g., the terminal or a byte stream over a network; for other streams, such as to files, (READP *FILE*) is equivalent to (NOT (EOFP *FILE*)).

> Note that because of line-buffering, READP may return T, indicating there is input in the buffer, but READ may still have to wait.

> Frequently, the terminal's input buffer contains a single EOL character left over from a previous input. For most applications, this situation wants to be treated as though the buffer were empty, and so READP returns NIL in this case. However, if *FLG*=T, READP returns T if there is *any* character in the input buffer, including a single EOL. *FLG* is ignored for streams other than the terminal.

(**EOFP** *FILE*)   [Function]

> Returns true if *FILE* is at "end of file", i.e., the next call to an input function would cause an END OF FILE error; NIL otherwise. For randomly accessible files, this can also be thought of as the file pointer pointing beyond the last byte of the file. *FILE* must be open for (at least) input, or an error is generated, FILE NOT OPEN.

> Note that EOFP can return NIL and yet the next call to READ might still cause an END OF FILE error, because the only characters remaining in the input were separators or otherwise constituted an incomplete expression. The function SKIPSEPRS is sometimes more useful as a way of detecting end of file when it is known that all the expressions in the file are well formed.

(**WAITFORINPUT** *FILE*)   [Function]

> Waits until input is available from *FILE* or from the terminal, i.e. from T. WAITFORINPUT is functionally equivalent to (until (OR (READP T) (READP FILE)) do NIL),

except that it does not use up machine cycles while waiting. Returns the device for which input is now available, i.e. *FILE* or T.

FILE can also be an integer, in which case WAITFORINPUT waits until there is input available from the terminal, or until *FILE* milliseconds have elapsed. Value is T if input is now available, NIL in the case that WAITFORINPUT timed out.

(**SKREAD** *FILE REREADSTRING RDTBL*) [Function]

"Skip Read". SKREAD consumes characters from *FILE* as if one call to READ had been performed, without paying the storage and compute cost to really read in the structure. *REREADSTRING* is for the case where the caller has already performed some READC's and RATOM's before deciding to skip this expression. In this case, *REREADSTRING* should be the material already read (as a string), and SKREAD operates as though it had seen that material first, thus setting up its parenthesis count, double-quote count, etc.

The read table *RDTBL* is used for reading from *FILE*. If *RDTBL* is NIL, it defaults to the value of FILERDTBL. SKREAD may have difficulties if unusual read macros are defined in *RDTBL*. SKREAD does not recognize read macro characters in *REREADSTRING*, nor SPLICE or INFIX read macros. This is only a problem if the read macros are defined to parse subsequent input in the stream that does not follow the normal parenthesis and string-quote conventions.

SKREAD returns %) if the read terminated on an unbalanced closing parenthesis; %] if the read terminated on an unbalanced %], i.e., one which also would have closed any extant open left parentheses; otherwise NIL.

(**SKIPSEPRS** *FILE RDTBL*) [Function]

Consumes characters from *FILE* until it encounters a non-separator character (as defined by *RDTBL*). SKIPSEPRS returns, but does not consume, the terminating character, so that the next call to READC would return the same character. If no non-separator character is found before the end of file is reached, SKIPSEPRS returns NIL and leaves the stream at end of file. This function is useful for skipping over "white space" when scanning a stream character by character, or for detecting end of file when reading expressions from a stream with no pre-arranged terminating expression.

Output Functions

Unless otherwise specified by DEFPRINT, pointers other than lists, strings, atoms, or numbers, are printed in the form {*DATATYPE*} followed by the octal representation of the address of the pointer (regardless of radix). For example, an array pointer might print as {ARRAYP}#43,2760. This printed representation is for compactness of display on your terminal, and will *not* read back in correctly; if the form above is read, it will produce the litatom {ARRAYP}#43,2760.

Note: The term "end-of-line" appearing in the description of an output function means the character or characters used to terminate a line in the file system being used

> by the given implementation of Interlisp. For example, in Interlisp-D end-of-line is indicated by the character carriage-return.

Some of the functions described below have a *RDTBL* argument, which specifies the read table to be used for output. If *RDTBL* is NIL, the primary read table is used.

Most of the functions described below have an argument *FILE*, which specifies the stream on which the operation is to take place. If *FILE* is NIL, the primary output stream is used .

(**OUTPUT** *FILE*)   [Function]

> Sets *FILE* as the primary output stream; returns the old primary output stream. *FILE* must be open for output.
>
> (OUTPUT) returns the current primary output stream, which is not changed.
>
> > Note: If the primary output stream is set to a file, the file's full name, rather than the stream itself, is returned. See the discussion in Chapter 24.

(**PRIN1** *X FILE*)   [Function]

> Prints *X* on *FILE*.

(**PRIN2** *X FILE RDTBL*)   [Function]

> Prints *X* on *FILE* with %'s and "'s inserted where required for it to read back in properly by READ, using *RDTBL*.

Both PRIN1 and PRIN2 print any kind of Lisp expression, including lists, atoms, numbers, and strings. PRIN1 is generally used for printing expressions where human readability, rather than machine readability, is important, e.g., when printing text rather than program fragments. PRIN1 does not print double quotes around strings, or % in front of special characters. PRIN2 is used for printing Interlisp expressions which can then be read back into Interlisp with READ; i.e., break and separator characters in atoms will be preceded by %'s. For example, the atom "()" is printed as %(%) by PRIN2. If the integer output radix (as set by RADIX) is not 10, PRIN2 prints the integer using the input syntax for non-decimal integers (see Chapter 7) but PRIN1 does not (but both print the integer in the output radix).

(**PRIN3** *X FILE*)   [Function]
(**PRIN4** *X FILE RDTBL*)   [Function]

> PRIN3 and PRIN4 are the same as PRIN1 and PRIN2 respectively, except that they do not increment the horizontal position counter nor perform any linelength checks. They are useful primarily for printing control characters.

(**PRINT** *X FILE RDTBL*)   [Function]

> Prints the expression *X* using PRIN2 followed by an end-of-line. Returns *X*.

(**PRINTCCODE** *CHARCODE FILE*)   [Function]

Outputs a single character whose code is *CHARCODE* to *FILE*. This is similar to (PRIN1
(CHARACTER *CHARCODE*)), except that numeric characters are guaranteed to print
"correctly"; e.g., (PRINTCCODE (CHARCODE 9)) always prints "9", independent of the
setting of RADIX.

PRINTCCODE may actually print more than one byte on *FILE*, due to character encoding
and end of line conventions; thus, no assumptions should be made about the relative
motion of the file pointer (see GETFILEPTR) during this operation.

(**BOUT** *STREAM BYTE*)   [Function]

Outputs a single 8-bit byte to *STREAM*. This is similar to PRINTCCODE, but for binary
streams the character position in *STREAM* is not updated (as with PRIN3), and end of line
conventions are ignored.

> Note: BOUT is similar to PRINTCCODE, except that BOUT always writes a
> single byte, whereas PRINTCCODE writes a "character" that can consist
> of more than one byte, depending on the character and its encoding.

(**SPACES** *N FILE*)   [Function]

Prints *N* spaces. Returns NIL.

(**TERPRI** *FILE*)   [Function]

Prints an end-of-line character. Returns NIL.

(**FRESHLINE** *STREAM*)   [Function]

Equivalent to TERPRI, except it does nothing if it is already at the beginning of the line.
Returns T if it prints an end-of-line, NIL otherwise.

(**TAB** *POS MINSPACES FILE*)   [Function]

Prints the appropriate number of spaces to move to position *POS*. *MINSPACES* indicates
how many spaces must be printed (if NIL, 1 is used). If the current position plus
MINSPACES is greater than *POS*, TAB does a TERPRI and then (SPACES *POS*). If
MINSPACES is T, and the current position is greater than *POS*, then TAB does nothing.

Note: A sequence of PRINT, PRIN2, SPACES, and TERPRI expressions can often be
more conveniently coded with a single PRINTOUT statement.

(**SHOWPRIN2** *X FILE RDTBL*)   [Function]

Like PRIN2 except if SYSPRETTYFLG=T, prettyprints *X* instead. Returns *X*.

(**SHOWPRINT** *X FILE RDTBL*)   [Function]

> Like PRINT except if SYSPRETTYFLG=T, prettyprints *X* instead, followed by an end-of-line. Returns *X*.

SHOWPRINT and SHOWPRIN2 are used by the programmer's assistant (Chapter 13) for printing the values of expressions and for printing the history list, by various commands of the break package (Chapter 14), e.g. ?= and BT commands, and various other system packages. The idea is that by simply settting or binding SYSPRETTYFLG to T (initially NIL), you instruct the system when interacting with you to PRETTYPRINT expressions (Chapter 26) instead of printing them.

(**PRINTBELLS**)   [Function]

> Used by DWIM (Chapter 19) to print a sequence of bells to alert you to stop typing. Can be advised or redefined for special applications, e.g., to flash the screen on a display terminal.

(**FORCEOUTPUT** *STREAM WAITFORFINISH*)   [Function]

> Forces any buffered output data in *STREAM* to be transmitted.

> If *WAITFORFINISH* is non-NIL, this doesn't return until the data has been forced out.

(**POSITION** *FILE N*)   [Function]

> Returns the column number at which the next character will be read or printed. After a end of line, the column number is 0. If *N* is non-NIL, *resets* the column number to be *N*.

> Note that resetting POSITION only changes Lisp's belief about the current column number; it does not cause any horizontal motion. Also note that (POSITION *FILE*) is *not* the same as (GETFILEPTR *FILE*) which gives the position in the *file*, not on the *line*.

(**LINELENGTH** *N FILE*)   [Function]

> Sets the length of the print line for the output file *FILE* to *N*; returns the former setting of the line length. *FILE* defaults to the primary output stream. (LINELENGTH NIL *FILE*) returns the current setting for *FILE*. When a file is first opened, its line length is set to the value of the variable FILELINELENGTH.

> Whenever printing an atom or string would increase a file's position *beyond* the line length of the file, an end of line is automatically inserted first. This action can be defeated by using PRIN3 and PRIN4.

(**SETLINELENGTH** *N*)   [Function]

> Sets the line length for the terminal by doing (LINELENGTH *N* T). If *N* is NIL, it determines *N* by consulting the operating system's belief about the terminal's characteristics. In Interlisp-D, this is a no-op.

PRINTLEVEL

When using Interlisp one often has to handle large, complicated lists, which are difficult to understand when printed out. `PRINTLEVEL` allows you to specify in how much detail lists should be printed. The print functions `PRINT`, `PRIN1`, and `PRIN2` are all affected by level parameters set by:

(**PRINTLEVEL** *CARVAL CDRVAL*) [Function]

> Sets the `CAR` print level to *CARVAL*, and the `CDR` print level to *CDRVAL*. Returns a list cell whose `CAR` and `CDR` are the old settings. `PRINTLEVEL` is initialized with the value (1000 . -1).

> In order that `PRINTLEVEL` can be used with `RESETFORM` or `RESETSAVE`, if *CARVAL* is a list cell it is equivalent to (PRINTLEVEL (CAR *CARVAL*)(CDR *CARVAL*)).

> (PRINTLEVEL *N* NIL) changes the `CAR` printlevel without affecting the `CDR` printlevel. (PRINTLEVEL NIL *N*) changes the `CDR` printlevel with affecting the `CAR` printlevel. (PRINTLEVEL) gives the current setting without changing either.

>> Note: Control-P (Chapter 30) can be used to change the `PRINTLEVEL` setting dynamically, even while Interlisp is printing.

The `CAR` printlevel specifies how "deep" to print a list. Specifically, it is the number of unpaired left parentheses which will be printed. Below that level, all lists will be printed as &. If the `CAR` printlevel is *negative*, the action is similar except that an end-of-line is inserted after each right parentheses that would be immediately followed by a left parenthesis.

The `CDR` printlevel specifies how "long" to print a list. It is the number of top level list elements that will be printed before the printing is terminated with --. For example, if *CDRVAL*=2, (A B C D E) will print as (A B --). For sublists, the number of list elements printed is also affected by the depth of printing in the `CAR` direction: Whenever the *sum* of the depth of the sublist (i.e. the number of unmatched left parentheses) and the number of elements is greater than the `CDR` printlevel, -- is printed. This gives a "triangular" effect in that less is printed the farther one goes in either `CAR` or `CDR` direction. If the `CDR` printlevel is negative, then it is the same as if the `CDR` printlevel were infinite.

Examples:

	After:	(A (B C (D (E F) G) H) K L) prints as:
(PRINTLEVEL 3 -1)		(A (B C (D & G) H) K L)
(PRINTLEVEL 2 -1)		(A (B C & H) K L)
(PRINTLEVEL 1 -1)		(A & K L)
(PRINTLEVEL 0 -1)		&
(PRINTLEVEL 1000 2)		(A (B --) --)
(PRINTLEVEL 1000 3)		(A (B C --) K --)

```
                    (PRINTLEVEL 1 3)    (A & K --)
```

PLVLFILEFLG   [Variable]

> Normally, PRINTLEVEL only affects terminal output. Output to all other files acts as though the print level is infinite. However, if PLVLFILEFLG is T (initially NIL), then PRINTLEVEL affects output to files as well.

The following three functions are useful for printing isolated expressions at a specified print level without going to the overhead of resetting the global print level.

(**LVLPRINT** *X FILE CARLVL CDRLVL TAIL*)   [Function]

> Performs PRINT of *X* to *FILE*, using as CAR and CDR print levels the values *CARLVL* and *CDRLVL*, respectively. Uses the T read table. If *TAIL* is specified, and *X* is a tail of it, then begins its printing with "...", rather than on open parenthesis.

(**LVLPRIN2** *X FILE CARLVL CDRLVL TAIL*)   [Function]

> Similar to LVLPRIN2, but performs a PRIN2.

(**LVLPRIN1** *X FILE CARLVL CDRLVL TAIL*)   [Function]

> Similar to LVLPRIN1, but performs a PRIN1.

Printing Numbers

How the ordinary printing functions (PRIN1, PRIN2, etc.) print numbers can be affected in several ways. RADIX influences the printing of integers, and FLTFMT influences the printing of floating point numbers. The setting of the variable PRXFLG determines how the symbol-manipulation functions handle numbers. The PRINTNUM package permits greater controls on the printed appearance of numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

(**RADIX** *N*)   [Function]

> Resets the output radix for integers to the absolute value of *N*. The value of RADIX is its previous setting. (RADIX) gives the current setting without changing it. The initial setting is 10.

> Note that RADIX affects output *only*. There is no input radix; on input, numbers are interpreted as decimal unless they are entered in a non-decimal radix with syntax such as 123Q, |b10101, |5r1234 (see Chapter 7). RADIX does not affect the behavior of UNPACK, etc., unless the value of PRXFLG (below) is T. For example, if PRXFLG is NIL and the radix is set to 8 with (RADIX 8), the value of (UNPACK 9) is (9), not (1 1).

> Using PRINTNUM (below) or the PRINTOUT command .I (below) is often a more convenient and appropriate way to print a single number in a specified radix than to globally change RADIX.

(**FLTFMT** *FORMAT*)   [Function]

> Resets the output format for floating point numbers to the FLOAT format *FORMAT* (see PRINTNUM below for a description of FLOAT formats). *FORMAT*=T specifies the default "free" formatting: some number of significant digits (a function of the implementation) are printed, with trailing zeros suppressed; numbers with sufficiently large or small exponents are instead printed in exponent notation.

> FLTFMT returns its current setting. (FLTFMT) returns the current setting without changing it. The initial setting is T.

> Note: In Interlisp-D, FLTFMT ignores the *WIDTH* and *PAD* fields of the format (they are implemented only by PRINTNUM).

Whether print name manipulation functions (UNPACK, NCHARS, etc.) use the values of RADIX and FLTFMT is determined by the variable PRXFLG:

PRXFLG   [Variable]

> If PRXFLG=NIL (the initial setting), then the "PRIN1" name used by PACK, UNPACK, MKSTRING, etc., is computed using base 10 for integers and the system default floating format for floating point numbers, independent of the current setting of RADIX or FLTFMT. If PRXFLG=T, then RADIX and FLTFMT do dictate the "PRIN1" name of numbers. Note that in this case, PACK and UNPACK are *not* inverses.

> Examples with (RADIX 8), (FLTFMT '(FLOAT 4 2)):

>> With PRXFLG=NIL,

>> ```
>> (UNPACK 13)   =>   (1 3)
>> (PACK '(A 9))  =>   A9
>> (UNPACK 1.2345)  =>  (1 %. 2 3 4 5)
>> ```

>> With PRXFLG=T,

>> ```
>> (UNPACK 13)   =>   (1 5)
>> (PACK '(A 9))  =>   A11
>> (UNPACK 1.2345)  =>  (1 %. 2 3)
>> ```

> Note that PRXFLG does not effect the radix of "PRIN2" names, so with (RADIX 8), (NCHARS 9 T), which uses PRIN2 names, would return 3, (since 9 would print as 11Q) for either setting of PRXFLG.

> Warning: Some system functions will not work correctly if PRXFLG is not NIL. Therefore, resetting the global value of PRXFLG is not recommended. It is much better to rebind PRXFLG as a SPECVAR for that part of a program where it needs to be non-NIL.

The basic function for printing numbers under format control is PRINTNUM. Its utility is considerably enhanced when used in conjunction with the PRINTOUT package, which implements a compact language for specifying complicated sequences of elementary printing operations, and makes fancy output formats easy to design and simple to program.

(**PRINTNUM** FORMAT NUMBER FILE) [Function]

Prints *NUMBER* on *FILE* according to the format *FORMAT*. *FORMAT* is a list structure with one of the forms described below.

If *FORMAT* is a list of the form (FIX *WIDTH RADIX PAD0 LEFTFLUSH*), this specifies a FIX format. *NUMBER* is rounded to the nearest integer, and then printed in a field *WIDTH* characters long with radix set to *RADIX* (or 10 if *RADIX*=NIL; note that the setting from the function RADIX is *not* used as the default). If *PAD0* and *LEFTFLUSH* are both NIL, the number is right-justified in the field, and the padding characters to the left of the leading digit are spaces. If *PAD0* is T, the character "0" is used for padding. If *LEFTFLUSH* is T, then the number is left-justified in the field, with trailing spaces to fill out *WIDTH* characters.

The following examples illustrate the effects of the FIX format options on the number 9 (the vertical bars indicate the field width):

FORMAT:	(PRINTNUM *FORMAT* 9) prints:
(FIX 2)	| 9|
(FIX 2 NIL T)	|09|
(FIX 12 8 T)	|000000000011|
(FIX 5 NIL NIL T)	|9 |

If *FORMAT* is a list of the form (FLOAT *WIDTH DECPART EXPPART PAD0 ROUND*), this specifies a FLOAT format. *NUMBER* is printed as a decimal number in a field *WIDTH* characters wide, with *DECPART* digits to the right of the decimal point. If *EXPPART* is not 0 (or NIL), the number is printed in exponent notation, with the exponent occupying *EXPPART* characters in the field. *EXPPART* should allow for the character E and an optional sign to be printed before the exponent digits. As with FIX format, padding on the left is with spaces, unless *PAD0* is T. If *ROUND* is given, it indicates the digit position at which rounding is to take place, counting from the leading digit of the number.

Interlisp-D interprets *WIDTH*=NIL to mean no padding, i.e., to use however much space the number needs, and interprets *DECPART*=NIL to mean as many decimal places as needed.

The following examples illustrate the effects of the FLOAT format options on the number 27.689 (the vertical bars indicate the field width):

FORMAT:	(PRINTNUM *FORMAT* 27.689) prints:
(FLOAT 7 2)	| 27.69|
(FLOAT 7 2 NIL 0)	|0027.69|
(FLOAT 7 2 2)	| 2.77E1|
(FLOAT 11 2 4)	| 2.77E+01|
(FLOAT 7 2 NIL NIL 1)	| 30.00|
(FLOAT 7 2 NIL NIL 2)	| 28.00|

NILNUMPRINTFLG [Variable]

> If `PRINTNUM`'s *NUMBER* argument is not a number and not `NIL`, a `NON-NUMERIC ARG` error is generated. If *NUMBER* is `NIL`, the effect depends on the setting of the variable `NILNUMPRINTFLG`. If `NILNUMPRINTFLG` is `NIL`, then the error occurs as usual. If it is non-`NIL`, then no error occurs, and the value of `NILNUMPRINTFLG` is printed right-justified in the field described by *FORMAT*. This option facilitates the printing of numbers in aggregates with missing values coded as `NIL`.

User Defined Printing

Initially, Interlisp only knows how to print in an interesting way objects of type litatom, number, string, list and stackp. All other types of objects are printed in the form {datatype} followed by the octal representation of the address of the pointer, a format that cannot be read back in to produce an equivalent object. When defining user data types (using the `DATATYPE` record type, Chapter 8), it is often desirable to specify as well how objects of that type should be printed, so as to make their contents readable, or at least more informative to the viewer. The function `DEFPRINT` is used to specify the printing format of a data type.

(**DEFPRINT** *TYPE FN*) [Function]

> *TYPE* is a type name. Whenever a printing function (`PRINT`, `PRIN1`, `PRIN2`, etc.) or a function requiring a print name (`CHCON`, `NCHARS`, etc.) encounters an object of the indicated type, *FN* is called with two arguments: the item to be printed and the name of the stream, if any, to which the object is to be printed. The second argument is `NIL` on calls that request the print name of an object without actually printing it.

> If *FN* returns a list of the form (*ITEM1* . *ITEM2*), *ITEM1* is printed using `PRIN1` (unless it is `NIL`), and then *ITEM2* is printed using `PRIN2` (unless it is `NIL`). No spaces are printed between the two items. Typically, *ITEM1* is a read macro character.

> If *FN* returns `NIL`, the datum is printed in the system default manner.

> If *FN* returns `T`, nothing further is printed; *FN* is assumed to have printed the object to the stream itself. Note that this case if permitted only when the second argument passed to *FN* is non-`NIL`; otherwise, there is no destination for *FN* to do its printing, so it must return as in one of the other two cases.

Printing Unusual Data Structures

`HPRINT` (for "Horrible Print") and `HREAD` provide a mechanism for printing and reading back in general data structures that cannot normally be dumped and loaded easily, such as (possibly re-entrant or circular) structures containing user datatypes, arrays, hash tables, as well as list structures. `HPRINT` will correctly print and read back in any structure containing any or all of the above, chasing all pointers down to the level of literal atoms, numbers or strings. `HPRINT` currently cannot handle compiled code arrays, stack positions, or arbitrary unboxed numbers.

HPRINT operates by simulating the Interlisp PRINT routine for normal list structures. When it encounters a user datatype (see Chapter 8), or an array or hash array, it prints the data contained therein, surrounded by special characters defined as read macro characters. While chasing the pointers of a structure, it also keeps a hash table of those items it encounters, and if any item is encountered a second time, another read macro character is inserted before the first occurrence (by resetting the file pointer with SETFILEPTR) and all subsequent occurrences are printed as a back reference using an appropriate macro character. Thus the inverse function, HREAD merely calls the Interlisp READ routine with the appropriate read table.

(**HPRINT** *EXPR FILE UNCIRCULAR DATATYPESEEN*)   [Function]

> Prints *EXPR* on *FILE*. If *UNCIRCULAR* is non-NIL, HPRINT does no checking for any circularities in *EXPR* (but is still useful for dumping arbitrary structures of arrays, hash arrays, lists, user data types, etc., that do not contain circularities). Specifying *UNCIRCULAR* as non-NIL results in a large speed and internal-storage advantage.

> Normally, when HPRINT encounters a user data type for the first time, it outputs a summary of the data type's declaration. When this is read in, the data type is redeclared. If *DATATYPESEEN* is non-NIL, HPRINT assumes that the same data type declarations will be in force at read time as were at HPRINT time, and not output declarations.

> HPRINT is intended primarily for output to random access files, since the algorithm depends on being able to reset the file pointer. If *FILE* is not a random access file (and *UNCIRCULAR* = NIL), a temporary file, HPRINT.SCRATCH, is opened, *EXPR* is HPRINTed on it, and then that file is copied to the final output file and the temporary file is deleted.

> You can not use HPRINT to save things that contains pointers to raw storage. Fontdescriptors contain pointers to raw storage and windows contain pointers to fontdescriptors. Netiher can therefor be saved with HPRINT.

(**HREAD** *FILE*)   [Function]

> Reads and returns an HPRINT-ed expression from *FILE*.

(**HCOPYALL** *X*)   [Function]

> Copies data structure *X*. *X* may contain circular pointers as well as arbitrary structures.

> Note: HORRIBLEVARS and UGLYVARS (Chapter 17) are two file package commands for dumping and reloading circular and re-entrant data structures. They provide a convenient interface to HPRINT and HREAD.

> When HPRINT is dumping a data structure that contains an instance of an Interlisp datatype, the datatype declaration is also printed onto the file. Reading such a data structure with HREAD can cause problems if it redefines a system datatype. Redefining a system datatype will almost definitely cause serious errors. The Interlisp system datatypes do not change very often, but there is always a possibility when loading in old files created under an old Interlisp release.

> To prevent accidental system crashes, HREAD will *not* redefine datatypes. Instead, it will cause an error "attempt   to   read   DATATYPE   with   different   field

`specification than currently defined`". Continuing from this error will redefine the datatype.

Random Access File Operations

For most applications, files are read starting at their beginning and proceeding sequentially, i.e., the next character read is the one immediately following the last character read. Similarly, files are written sequentially. However, for files on some devices, it is also possible to read/write characters at arbitrary positions in a file, essentially treating the file as a large block of auxiliary storage. For example, one application might involve writing an expression at the *beginning* of the file, and then reading an expression from a specified point in its *middle*. This particular example requires the file be open for *both* input and output. However, random file input or output can also be performed on files that have been opened for only input or only output.

Associated with each file is a "file pointer" that points to the location where the next character is to be read from or written to. The file position of a byte is the number of bytes that precede it in the file, i.e., 0 is the position of the beginning of the file. The file pointer to a file is automatically advanced after each input or output operation. This section describes functions which can be used to *reposition* the file pointer on those files that can be randomly accessed. A file used in this fashion is much like an array in that it has a certain number of addressable locations that characters can be put into or taken from. However, unlike arrays, files can be enlarged. For example, if the file pointer is positioned at the end of a file and anything is written, the file "grows." It is also possible to position the file pointer *beyond* the end of file and then to write. (If the program attempts to *read* beyond the end of file, an END OF FILE error occurs.) In this case, the file is enlarged, and a "hole" is created, which can later be written into. Note that this enlargement only takes place at the *end* of a file; it is not possible to make more room in the middle of a file. In other words, if expression A begins at position 1000, and expression B at 1100, and the program attempts to overwrite A with expression C, whose printed representation is 200 bytes long, part of B will be altered.

Warning: File positions are always in terms of bytes, not characters. You should thus be very careful about computing the space needed for an expression. In particular, NS characters may take multiple bytes (see below). Also, the end-of-line character (see Chapter 24) may be represented by a different number of characters in different implementations. Output functions may also introduce end-of-line's as a result of LINELENGTH considerations. Therefore NCHARS (see Chapter 2) does *not* specify how many bytes an expression takes to print, even ignoring line length considerations.

(**GETFILEPTR** *FILE*)   [Function]

Returns the current position of the file pointer for *FILE*, i.e., the byte address at which the next input/output operation will commence.

(**SETFILEPTR** *FILE ADR*)   [Function]

Sets the file pointer for *FILE* to the position *ADR*; returns *ADR*. The special value *ADR=-1* is interpreted to mean the address of the end of file.

Note: If a file is opened for output only, the end of file is initially zero, even if an old file by the same name had existed (see OPENSTREAM, Chapter 24). If a file is opened for both input and output, the initial file pointer is the beginning of the file, but (SETFILEPTR *FILE* –1) sets it to the end of the file. If the file had been opened in append mode by (OPENSTREAM *FILE* 'APPEND), the file pointer right after opening would be set to the end of the existing file, in which case a SETFILEPTR to position the file at the end would be unnecessary.

(**GETEOFPTR** *FILE*) [Function]

Returns the byte address of the end of file, i.e., the number of bytes in the file. Equivalent to performing (SETFILEPTR *FILE* –1) and returning (GETFILEPTR *FILE*) except that it does not change the current file pointer.

(**RANDACCESSP** *FILE*) [Function]

Returns *FILE* if *FILE* is randomly accessible, NIL otherwise. The file T is not randomly accessible, nor are certain network file connections in Interlisp-D. *FILE* must be open or an error is generated, FILE NOT OPEN.

(**COPYBYTES** *SRCFIL DSTFIL START END*) [Function]

Copies bytes from *SRCFIL* to *DSTFIL*, starting from position *START* and up to but not including position *END*. Both *SRCFIL* and *DSTFIL* must be open. Returns T.

If *END*=NIL, *START* is interpreted as the number of bytes to copy (starting at the current position). If *START* is also NIL, bytes are copied until the end of the file is reached.

Warning: COPYBYTES does not take any account of multi-byte NS characters (see Chapter 2). COPYCHARS (below) should be used whenever copying information that might include NS characters.

(**COPYCHARS** *SRCFIL DSTFIL START END*) [Function]

Like COPYBYTES except that it copies NS characters (see Chapter 2), and performs the proper conversion if the end-of-line conventions of *SRCFIL* and *DSTFIL* are not the same (see Chapter 24). *START* and *END* are interpreted the same as with COPYBYTES, i.e., as byte (not character) specifications in *SRCFIL*. The number of bytes actually output to *DSTFIL* might be more or less than the number of bytes specified by *START* and *END*, depending on what the end-of-line conventions are. In the case where the end-of-line conventions happen to be the same, COPYCHARS simply calls COPYBYTES.

(**FILEPOS** *STR FILE START END SKIP TAIL CASEARRAY*) [Function]

Analogous to STRPOS (see Chapter 4), but searches a file rather than a string. FILEPOS searches *FILE* for the string *STR*. Search begins at *START* (or the current position of the file pointer, if *START*=NIL), and goes to *END* (or the end of *FILE*, if *END*=NIL). Returns the address of the start of the match, or NIL if not found.

SKIP can be used to specify a character which matches any character in the file. If *TAIL* is T, and the search is successful, the value is the address of the first character *after* the sequence of characters corresponding to *STR*, instead of the starting address of the sequence. In either case, the file is left so that the next i/o operation begins at the address returned as the value of FILEPOS.

CASEARRAY should be a "case array" that specifies that certain characters should be transformed to other characters before matching. Case arrays are returned by CASEARRAY or SEPRCASE below. *CASEARRAY*=NIL means no transformation will be performed.

A case array is an implementation-dependent object that is logically an array of character codes with one entry for each possible character. FILEPOS maps each character in the file "through" *CASEARRAY* in the sense that each character code is transformed into the corresponding character code from *CASEARRAY* before matching. Thus if two characters map into the same value, they are treated as equivalent by FILEPOS. CASEARRAY and SETCASEARRAY provide an implementation-independent interface to case arrays.

For example, to search without regard to upper and lower case differences, *CASEARRAY* would be a case array where all characters map to themselves, except for lower case characters, whose corresponding elements would be the upper case characters. To search for a delimited atom, one could use " *ATOM* " as the pattern, and specify a case array in which all of the break and separator characters mapped into the same code as space.

For applications calling for extensive file searches, the function FFILEPOS is often faster than FILEPOS.

(**FFILEPOS** *PATTERN FILE START END SKIP TAIL CASEARRAY*)   [Function]

Like FILEPOS, except much faster in most applications. FFILEPOS is an implementation of the Boyer-Moore fast string searching algorithm. This algorithm preprocesses the string being searched for and then scans through the file in steps usually equal to the length of the string. Thus, FFILEPOS speeds up roughly in proportion to the length of the string, e.g., a string of length 10 will be found twice as fast as a string of length 5 in the same position.

Because of certain fixed overheads, it is generally better to use FILEPOS for short searches or short strings.

(**CASEARRAY** *OLDARRAY*)   [Function]

Creates and returns a new case array, with all elements set to themselves, to indicate the identity mapping. If *OLDARRAY* is given, it is reused.

(**SETCASEARRAY** *CASEARRAY FROMCODE TOCODE*)   [Function]

Modifies the case array *CASEARRAY* so that character code *FROMCODE* is mapped to character code *TOCODE*.

(**GETCASEARRAY** *CASEARRAY FROMCODE*)   [Function]

> Returns the character code that *FROMCODE* is mapped to in *CASEARRAY*.

(**SEPRCASE** *CLFLG*)   [Function]

> Returns a new case array suitable for use by FILEPOS or FFILEPOS in which all of the break/separators of FILERDTBL are mapped into character code zero. If *CLFLG* is non-NIL, then all CLISP characters are mapped into this character as well. This is useful for finding a delimited atom in a file. For example, if *PATTERN* is " FOO ", and (SEPRCASE T) is used for *CASEARRAY*, then FILEPOS will find "(FOO_".

UPPERCASEARRAY   [Variable]

> Value is a case array in which every lowercase character is mapped into the corresponding uppercase character. Useful for searching text files.

Input/Output Operations with Characters and Bytes

Interlisp-D supports the 16-bit NS character set (see Chapter 2). All of the standard string and print name functions accept litatoms and strings containing NS characters. In almost all cases, a program does not have to distinguish between NS characters or 8-bit characters. The exception to this rule is the handling of input/output operations.

Interlisp-D uses two ways of writing 16-bit NS characters on files. One way is to write the full 16-bits (two bytes) every time a character is output. The other way is to use "run-encoding." Each 16 NS character can be decoded into a character set (an integer from 0 to 254 inclusive) and a character number (also an integer from 0 to 254 inclusive). In run-encoding, the byte 255 (illegal as either a character set number or a character number) is used to signal a change to a given character set, and the following bytes are all assumed to come from the same character set (until the next change-character set sequence). Run-encoding can reduce the number of bytes required to encode a string of NS characters, as long as there are long sequences of characters from the same character set (usually the case).

Note that characters are not the same as bytes. A single character can take anywhere from one to four bytes bytes, depending on whether it is in the same character set as the preceeding character, and whether run-encoding is enabled. Programs which assume that characters are equal to bytes must be changed to work with NS characters.

The functions BIN and BOUT (see above) should only be used to read and write single eight-bit bytes. The functions READCCODE and PRINTCCODE (see above) should be used to read and write single character codes, interpreting run-encoded NS characters. COPYBYTES should only be used to copy blocks of 8-bit data; COPYCHARS should be used to copy characters. Most I/O functions (READC, PRIN1, etc.) read or write 16-bit NS characters.

The use of NS characters has serious consequences for any program that uses file pointers to access a file in a random access manner. At any point when a file is being read or written, it has a "current character set." If the file pointer is changed with SETFILEPTR to a part of the file with a different character set, any characters read or written may have the wrong character set. The current character set can be accessed with the following function:

(**CHARSET** *STREAM CHARACTERSET*)   [Function]

> Returns the current character set of the stream *STREAM*. If *CHARACTERSET* is non-NIL, the current character set for *STREAM* is set. Note that for output streams this may cause bytes to be written to the stream.
>
> If *CHARACTERSET* is T, run encoding for *STREAM* is disabled: both the character set and the character number (two bytes total) will be written to the stream for each character printed.

PRINTOUT

Interlisp provides many facilities for controlling the format of printed output. By executing various sequences of PRIN1, PRIN2, TAB, TERPRI, SPACES, PRINTNUM, and PRINTDEF, almost any effect can be achieved. PRINTOUT implements a compact language for specifying complicated sequences of these elementary printing functions. It makes fancy output formats easy to design and simple to program.

PRINTOUT is a CLISP word (like FOR and IF) for interpreting a special printing language in which you can describe the kinds of printing desired. The description is translated by DWIMIFY to the appropriate sequence of PRIN1, TAB, etc., before it is evaluated or compiled. PRINTOUT printing descriptions have the following general form:

$$(\text{PRINTOUT } STREAM \ PRINTCOM_1 \ \dots \ PRINTCOM_N)$$

STREAM is evaluated to obtain the stream to which the output from this specification is directed. The PRINTOUT commands are strung together, one after the other without punctuation, after *STREAM*. Some commands occupy a single position in this list, but many commands expect to find arguments following the command name in the list. The commands fall into several logical groups: one set deals with horizontal and vertical spacing, another group provides controls for certain formatting capabilities (font changes and subscripting), while a third set is concerned with various ways of actually printing items. Finally, there is a command that permits escaping to a simple Lisp evaluation in the middle of a PRINTOUT form. The various commands are described below. The following examples give a general flavor of how PRINTOUT is used:

Example 1: Suppose you want to print out on the terminal the values of three variables, X, Y, and Z, separated by spaces and followed by a carriage return. This could be done by:

```
(PRIN1 X T)
        (SPACES 1 T)
        (PRIN1 Y T)
        (SPACES 1 T)
        (PRIN1 Z T)
        (TERPRI T)
```

or by the more concise PRINTOUT form:

```
(PRINTOUT T X , Y , Z T)
```

Here the first T specifies output to the terminal, the commas cause single spaces to be printed, and the final T specifies a TERPRI. The variable names are not recognized as special PRINTOUT commands, so they are printed using PRIN1 by default.

Example 2: Suppose the values of X and Y are to be pretty-printed lined up at position 10, preceded by identifying strings. If the output is to go to the primary output stream, you could write either:

```
(PRIN1 "X =")
        (PRINTDEF X 10 T)
        (TERPRI )
        (PRIN1 "Y =")
        (PRINTDEF Y 10 T)
        (TERPRI)
```

or the equivalent:

```
(PRINTOUT NIL "X =" 10 .PPV X T
          "Y =" 10 .PPV Y T)
```

Since strings are not recognized as special commands, "X =" is also printed with PRIN1 by default. The positive integer means TAB to position 10, where the .PPV command causes the value of X to be prettyprinted as a variable. By convention, special atoms used as PRINTOUT commands are prefixed with a period. The T causes a carriage return, so the Y information is printed on the next line.

Example 3. As a final example, suppose that the value of X is an integer and the value of Y is a floating-point number. X is to be printed right-flushed in a field of width 5 beginning at position 15, and Y is to be printed in a field of width 10 also starting at position 15 with 2 places to the right of the decimal point. Furthermore, suppose that the variable names are to appear in the font class named BOLDFONT and the values in font class SMALLFONT. The program in ordinary Interlisp that would accomplish these effects is too complicated to include here. With PRINTOUT, one could write:

```
(PRINTOUT NIL
          .FONT BOLDFONT "X =" 15
```

```
                     .FONT SMALLFONT .I5 X T
                     .FONT BOLDFONT "Y =" 15
                     .FONT SMALLFONT .F10.2 Y T
                     .FONT BOLDFONT)
```

The `.FONT` commands do whatever is necessary to change the font on a multi-font output device. The `.I5` command sets up a `FIX` format for a call to the function `PRINTNUM` (see above) to print `X` in the desired format. The `.F10.2` specifies a `FLOAT` format for `PRINTNUM`.

Horizontal Spacing Commands

The horizontal spacing commands provide convenient ways of calling `TAB` and `SPACES`. In the following descriptions, *N* stands for a literal positive integer (*not* for a variable or expression whose value is an integer).

N (*N* a number)   [PRINTOUT Command]

> Used for absolute spacing. It results in a `TAB` to position *N* (literally, a (`TAB N`)). If the line is currently at position *N* or beyond, the file will be positioned at position *N* on the next line.

.TAB *POS*   [PRINTOUT Command]

> Specifies `TAB` to position (the value of) *POS*. This is one of several commands whose effect could be achieved by simply escaping to Lisp, and executing the corresponding form. It is provided as a separate command so that the `PRINTOUT` form is more concise and is prettyprinted more compactly. Note that `.TAB` *N* and *N*, where *N* is an integer, are equivalent.

.TAB0 *POS*   [PRINTOUT Command]

> Like `.TAB` except that it can result in zero spaces (i.e. the call to `TAB` specifies *MINSPACES*=0).

–N (*N* a number)   [PRINTOUT Command]

> Negative integers indicate relative (as opposed to absolute) spacing. Translates as (`SPACES` $|N|$).

,   [PRINTOUT Command]
,,   [PRINTOUT Command]
,,,   [PRINTOUT Command]

> (1, 2 or 3 commas) Provides a short-hand way of specifying 1, 2 or 3 spaces, i.e., these commands are equivalent to –1, –2, and –3, respectively.

.SP *DISTANCE*   [PRINTOUT Command]

> Translates as (`SPACES` *DISTANCE*). Note that `.SP` *N* and –*N*, where *N* is an integer, are equivalent.

Vertical Spacing Commands

Vertical spacing is obtained by calling TERPRI or printing form-feeds. The relevant commands are:

T   [PRINTOUT Command]

> Translates as (TERPRI), i.e., move to position 0 (the first column) of the next line. To print the letter T, use the string "T".

.SKIP *LINES*   [PRINTOUT Command]

> Equivalent to a sequence of *LINES* (TERPRI)'s. The .SKIP command allows for skipping large constant distances and for computing the distance to be skipped.

.PAGE   [PRINTOUT Command]

> Puts a form-feed (Control-L) out on the file. Care is taken to make sure that Interlisp's view of the current line position is correctly updated.

Special Formatting Controls

There are a small number of commands for invoking some of the formatting capabilities of multi-font output devices. The available commands are:

.FONT *FONTSPEC*   [PRINTOUT Command]

> Changes printing to the font *FONTSPEC*, which can be a font descriptor, a "font list" such as '(MODERN 10), an image stream (coerced to its current font), or a windows (coerced to the current font of its display stream). The DSPFONT is changed permanently. See fonts (Chapter 27) for more information.

> *FONTSPEC* may also be a positive integer *N*, which is taken as an abbreviated reference to the font class named FONT*N* (e.g. 1 => FONT1).

.SUP   [PRINTOUT Command]

> Specifies superscripting. All subsequent characters are printed above the base of the current line. Note that this is absolute, not relative: a .SUP following a .SUP is a no-op.

.SUB   [PRINTOUT Command]

> Specifies subscripting. Subsequent printing is below the base of the current line. As with superscripting, the effect is absolute.

.BASE   [PRINTOUT Command]

> Moves printing back to the base of the current line. Un-does a previous .SUP or .SUB; a no-op, if printing is currently at the base.

Printing Specifications

The value of any expression in a PRINTOUT form that is not recognized as a command itself or as a command argument is printed using PRIN1 by default. For example, title strings can be printed by simply including the string as a separate PRINTOUT command, and the values of variables and forms can be printed in much the same way. Note that a literal integer, say 51, cannot be printed by including it as a command, since it would be interpreted as a TAB; the desired effect can be obtained by using instead the string specification "51", or the form (QUOTE 51).

For those instances when PRIN1 is not appropriate, e.g., PRIN2 is required, or a list structures must be prettyprinted, the following commands are available:

.P2 *THING* [PRINTOUT Command]

Causes *THING* to be printed using PRIN2; translates as (PRIN2 *THING*).

.PPF *THING* [PRINTOUT Command]

Causes *THING* to be prettyprinted at the current line position via PRINTDEF (see Chapter 26). The call to PRINTDEF specifies that *THING* is to be printed as if it were part of a function definition. That is, SELECTQ, PROG, etc., receive special treatment.

.PPV *THING* [PRINTOUT Command]

Prettyprints *THING* as a variable; no special interpretation is given to SELECTQ, PROG, etc.

.PPFTL *THING* [PRINTOUT Command]

Like .PPF, but prettyprints *THING* as a *tail*, that is, without the initial and final parentheses if it is a list. Useful for prettyprinting sub-lists of a list whose other elements are formatted with other commands.

.PPVTL *THING* [PRINTOUT Command]

Like .PPV, but prettyprints *THING* as a tail.

Paragraph Format

Interlisp's prettyprint routines are designed to display the structure of expressions, but they are not really suitable for formatting unstructured text. If a list is to be printed as a textual paragraph, its internal structure is less important than controlling its left and right margins, and the indentation of its first line. The .PARA and .PARA2 commands allow these parameters to be conveniently specified.

.PARA *LMARG RMARG LIST* [PRINTOUT Command]

Prints *LIST* in paragraph format, using PRIN1. Translates as (PRINTPARA *LMARG RMARG LIST*) (see below).

Example: `(PRINTOUT T 10 .PARA 5 -5 LST)` will print the elements of `LST` as a paragraph with left margin at 5, right margin at `(LINELENGTH)`-5, and the first line indented to 10.

.PARA2 *LMARG RMARG LIST*   [PRINTOUT Command]

Print as paragraph using `PRIN2` instead of `PRIN1`. Translates as (`PRINTPARA` *LMARG RMARG LIST* T).

Right-Flushing

Two commands are provided for printing simple expressions flushed-right against a specified line position, using the function `FLUSHRIGHT` (see below). They take into account the current position, the number of characters in the print-name of the expression, and the position the expression is to be flush against, and then print the appropriate number of spaces to achieve the desired effect. Note that this might entail going to a new line before printing. Note also that right-flushing of expressions longer than a line (e.g. a large list) makes little sense, and the appearance of the output is not guaranteed.

.FR *POS EXPR*   [PRINTOUT Command]

Flush-right using `PRIN1`. The value of *POS* determines the position that the right end of *EXPR* will line up at. As with the horizontal spacing commands, a negative position number means |*POS*| columns from the current position, a positive number specifies the position absolutely. *POS*=0 specifies the right-margin, i.e. is interpreted as (`LINELENGTH`).

.FR2 *POS EXPR*   [PRINTOUT Command]

Flush-right using `PRIN2` instead of `PRIN1`.

Centering

Commands for centering simple expressions between the current line position and another specified position are also available. As with right flushing, centering of large expressions is not guaranteed.

.CENTER *POS EXPR*   [PRINTOUT Command]

Centers *EXPR* between the current line position and the position specified by the value of *POS*. A positive *POS* is an absolute position number, a negative *POS* specifies a position relative to the current position, and 0 indicates the right-margin. Uses `PRIN1` for printing.

.CENTER2 *POS EXPR*   [PRINTOUT Command]

Centers using `PRIN2` instead of `PRIN1`.

Numbering

The following commands provide FORTRAN-like formatting capabilities for integer and floating-point numbers. Each command specifies a printing format and a number to be printed. The format specification translates into a format-list for the function PRINTNUM.

.I*FORMAT NUMBER*   [PRINTOUT Command]

> Specifies integer printing. Translates as a call to the function PRINTNUM with a FIX format-list constructed from *FORMAT*. The atomic format is broken apart at internal periods to form the format-list. For example, .I5.8.T yields the format-list (FIX 5 8 T), and the command sequence (PRINTOUT T .I5.8.T FOO) translates as (PRINTNUM '(FIX 5 8 T) FOO). This expression causes the value of FOO to be printed in radix 8 right-flushed in a field of width 5, with 0's used for padding on the left. Internal NIL's in the format specification may be omitted, e.g., the commands .I5**.**T and .I5.NIL.T are equivalent.

> The format specification .I1 is often useful for forcing a number to be printed in radix 10 (but not otherwise specially formatted), independent of the current setting of RADIX.

.F *FORMAT NUMBER*   [PRINTOUT Command]

> Specifies floating-number printing. Like the .I format command, except translates with a FLOAT format-list.

.N *FORMAT NUMBER*   [PRINTOUT Command]

> The .I and .F commands specify calls to PRINTNUM with quoted format specifications. The .N command translates as (PRINTNUM *FORMAT NUMBER*), i.e., it permits the format to be the value of some expression. Note that, unlike the .I and .F commands, *FORMAT* is a separate element in the command list, not part of an atom beginning with .N.

Escaping to Lisp

There are many reasons for taking control away from PRINTOUT in the middle of a long printing expression. Common situations involve temporary changes to system printing parameters (e.g. LINELENGTH), conditional printing (e.g. print FOO only if FIE is T), or lower-level iterative printing within a higher-level print specification.

FORM   [PRINTOUT Command]

> The escape command. *FORM* is an arbitrary Lisp expression that is evaluated within the context established by the PRINTOUT form, i.e., *FORM* can assume that the primary output stream has been set to be the *FILE* argument to PRINTOUT. Note that nothing is done with the *value* of *FORM*; any printing desired is accomplished by *FORM* itself, and the value is discarded.

> Note: Although PRINTOUT logically encloses its translation in a RESETFORM (Chapter 14) to change the primary output file to the *FILE* argument (if non-NIL), in most

cases it can actually pass *FILE* (or a locally bound variable if *FILE* is a non-trivial expression) to each printing function. Thus, the RESETFORM is only generated when the # command is used, or user-defined commands (below) are used. If many such occur in repeated PRINTOUT forms, it may be more efficient to embed them all in a single RESETFORM which changes the primary output file, and then specify *FILE*=NIL in the PRINTOUT expressions themselves.

User-Defined Commands

The collection of commands and options outlined above is aimed at fulfilling all common printing needs. However, certain applications might have other, more specialized printing idioms, so a facility is provided whereby you can define new commands. This is done by adding entries to the global list PRINTOUTMACROS to define how the new commands are to be translated.

PRINTOUTMACROS [Variable]

PRINTOUTMACROS is an association-list whose elements are of the form (*COMM* *FN*). Whenever *COMM* appears in command position in the sequence of PRINTOUT commands (as opposed to an argument position of another command), *FN* is applied to the tail of the command-list (including the command).

After inspecting as much of the tail as necessary, the function must return a list whose CAR is the translation of the user-defined command and its arguments, and whose CDR is the list of commands still remaining to be translated in the normal way.

For example, suppose you want to define a command "?", which will cause its single argument to be printed with PRIN1 only if it is not NIL. This can be done by entering (? ?TRAN) on PRINTOUTMACROS, and defining the function ?TRAN as follows:

```
(DEFINEQ (?TRAN (COMS)
   (CONS
      (SUBST (CADR COMS) 'ARG
         '(PROG ((TEMP ARG))
               (COND (TEMP (PRIN1 TEMP)))))
      (CDDR COMS))]
```

Note that ?TRAN does not do any printing itself; it returns a form which, when evaluated in the proper context, will perform the desired action. This form should direct all printing to the primary output file.

Special Printing Functions

The paragraph printing commands are translated into calls on the function PRINTPARA, which may also be called directly:

(**PRINTPARA** *LMARG RMARG LIST P2FLAG PARENFLAG FILE*) [Function]

Prints *LIST* on *FILE* in line-filled paragraph format with its first element beginning at the current line position and ending at or before *RMARG*, and with subsequent lines appearing

between *LMARG* and *RMARG*. If *P2FLAG* is non-NIL, prints elements using PRIN2, otherwise PRIN1. If *PARENFLAG* is non-NIL, then parentheses will be printed around the elements of *LIST*.

If *LMARG* is zero or positive, it is interpreted as an absolute column position. If it is negative, then the left margin will be at |*LMARG*|+(POSITION). If *LMARG*=NIL, the left margin will be at (POSITION), and the paragraph will appear in block format.

If *RMARG* is positive, it also is an absolute column position (which may be greater than the current (LINELENGTH)). Otherwise, it is interpreted as relative to (LINELENGTH), i.e., the right margin will be at (LINELENGTH)+|*RMARG*|. Example: (TAB 10) (PRINTPARA 5 –5 LST T) will PRIN2 the elements of LST in a paragraph with the first line beginning at column 10, subsequent lines beginning at column 5, and all lines ending at or before (LINELENGTH)–5.

The current (LINELENGTH) is unaffected by PRINTPARA, and upon completion, *FILE* will be positioned immediately after the last character of the last item of *LIST*. PRINTPARA is a no-op if *LIST* is not a list.

The right-flushing and centering commands translate as calls to the function FLUSHRIGHT:

(**FLUSHRIGHT** *POS X MIN P2FLAG CENTERFLAG FILE*) [Function]

If *CENTERFLAG*=NIL, prints *X* right-flushed against position *POS* on *FILE*; otherwise, centers *X* between the current line position and *POS*. Makes sure that it spaces over at least *MIN* spaces before printing by doing a TERPRI if necessary; *MIN*=NIL is equivalent to *MIN*=1. A positive *POS* indicates an absolute position, while a negative *POS* signifies the position which is |*POS*| to the right of the current line position. *POS*=0 is interpreted as (LINELENGTH), the right margin.

READFILE and WRITEFILE

For those applications where you simply want to simply read all of the expressions on a file, and not evaluate them, the function READFILE is available:

(**READFILE** *FILE RDTBL ENDTOKEN*) [NoSpread Function]

Reads successive expressions from file using READ (with read table *RDTBL*) until the single litatom *ENDTOKEN* is read, or an end of file encountered. Returns a list of these expressions.

If *RDTBL* is not specified, it defaults to FILERDTBL. If *ENDTOKEN* is not specified, it defaults to the litatom STOP.

(**WRITEFILE** *X FILE*) [Function]

Writes a date expression onto *FILE*, followed by successive expressions from *X*, using FILERDTBL as a read table. If *X* is atomic, its value is used. If *FILE* is not open, it is

opened. If *FILE* is a list, (CAR *FILE*) is used and the file is left opened. Otherwise, when *X* is finished, the litatom STOP is printed on *FILE* and it is closed. Returns *FILE*.

(**ENDFILE** *FILE*)   [Function]

> Prints STOP on *FILE* and closes it.

Read Tables

Many Interlisp input functions treat certain characters in special ways. For example, READ recognizes that the right and left parenthesis characters are used to specify list structures, and that the quote character is used to delimit text strings. The Interlisp input and (to a certain extent) output routines are table driven by read tables. Read tables are objects that specify the syntactic properties of characters for input routines. Since the input routines parse character sequences into objects, the read table in use determines which sequences are recognized as literal atoms, strings, list structures, etc.

Most Interlisp input functions take an optional read table argument, which specifies the read table to use when reading an expression. If NIL is given as the read table, the "primary read table" is used. If T is specified, the system terminal read table is used. Some functions will also accept the atom ORIG (*not* the *value* of ORIG) as indicating the "original" system read table. Some output functions also take a read table argument. For example, PRIN2 prints an expression so that it would be read in correctly using a given read table.

The Interlisp-D system uses the following read tables: T for input/output from terminals, the value of FILERDTBL for input/output from files, the value of EDITRDTBL for input from terminals while in the tty-based editor, the value of DEDITRDTBL for input from terminals while in the display-based editor, and the value of CODERDTBL for input/output from compiled files. These five read tables are initially copies of the ORIG read table, with changes made to some of them to provide read macros that are specific to terminal input or file input. Using the functions described below, you may further change, reset, or copy these tables. However, in the case of FILERDTBL and CODERDTBL, you are cautioned that changing these tables may prevent the system from being able to read files made with the original tables, or prevent users possessing only the standard tables from reading files made using the modified tables.

You can also create new read tables, and either explicitly pass them to input/output functions as arguments, or install them as the primary read table, via SETREADTABLE, and then not specify a *RDTBL* argument, i.e., use NIL.

Read Table Functions

(**READTABLEP** *RDTBL*)   [Function]

> Returns *RDTBL* if *RDTBL* is a real read table (*not* T or ORIG), otherwise NIL.

(**GETREADTABLE** *RDTBL*)   [Function]

> If *RDTBL*=NIL, returns the primary read table. If *RDTBL*=T, returns the system terminal
> read table. If *RDTBL* is a real read table, returns *RDTBL*. Otherwise, generates an ILLEGAL
> READTABLE error.

(**SETREADTABLE** *RDTBL FLG*)   [Function]

> Sets the primary read table to *RDTBL*. If *FLG*=T, SETREADTABLE sets the system terminal
> read table, T. Note that you can reset the other system read tables with SETQ, e.g., (SETQ
> FILERDTBL (GETREADTABLE)).

> Generates an ILLEGAL READTABLE error if *RDTBL* is not NIL, T, or a real read table.
> Returns the previous setting of the primary read table, so SETREADTABLE is suitable for
> use with RESETFORM (Chapter 14).

(**COPYREADTABLE** *RDTBL*)   [Function]

> Returns a copy of *RDTBL*. *RDTBL* can be a real read table, NIL, T, or ORIG (in which case
> COPYREADTABLE returns a copy of the *original* system read table), otherwise
> COPYREADTABLE generates an ILLEGAL READTABLE error.

> Note that COPYREADTABLE is the only function that *creates* a read table.

(**RESETREADTABLE** *RDTBL FROM*)   [Function]

> Copies (smashes) *FROM* into *RDTBL*. *FROM* and *RDTBL* can be NIL, T, or a real read table.
> In addition, *FROM* can be ORIG, meaning use the system's original read table.

Syntax Classes

A read table is an object that contains information about the "syntax class" of each character. There are
nine basic syntax classes: LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET,
STRINGDELIM, ESCAPE, BREAKCHAR, SEPRCHAR, and OTHER, each associated with a primitive
syntactic property. In addition, there is an unlimited assortment of user-defined syntax classes,
known as "read macros". The basic syntax classes are interpreted as follows:

> LEFTPAREN   (normally left parenthesis) Begins list structure.

> RIGHTPAREN   (normally right parenthesis) Ends list structure.

> LEFTBRACKET   (normally left bracket) Begins list structure. Also matches RIGHTBRACKET
> characters.

> RIGHTBRACKET   (normally left bracket) Ends list structure. Can close an arbitrary numbers of
> LEFTPAREN lists, back to the last LEFTBRACKET.

> STRINGDELIM   (normally double quote) Begins and ends text strings. Within the string, all
> characters except for the one(s) with class ESCAPE are treated as ordinary, i.e.,
> interpreted as if they were of syntax class OTHER. To include the string
> delimiter inside a string, prefix it with the ESCAPE character.

ESCAPE	(normally percent sign) Inhibits any special interpretation of the next character, i.e., the next character is interpreted to be of class OTHER, independent of its normal syntax class.
BREAKCHAR	(None initially) Is a break character, i.e., delimits atoms, but is otherwise an ordinary character.
SEPRCHAR	(space, carriage return, etc.) Delimits atoms, and is otherwise ignored.
OTHER	Characters that are not otherwise special belong to the class OTHER.

Characters of syntax class LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, and STRINGDELIM are all *break* characters. That is, in addition to their interpretation as delimiting list or string structures, they also terminate the reading of an atom. Characters of class BREAKCHAR serve *only* to terminate atoms, with no other special meaning. In addition, if a break character is the first non-separator encountered by RATOM, it is read as a one-character atom. In order for a break character to be included in an atom, it must be preceded by the ESCAPE character.

Characters of class SEPRCHAR also terminate atoms, but are otherwise completely ignored; they can be thought of as logically spaces. As with break characters, they must be preceded by the ESCAPE character in order to appear in an atom.

For example, if $ were a break character and * a separator character, the input stream ABC**DEF$GH*$$ would be read by six calls to RATOM returning respectively ABC, DEF, $, GH, $, $.

Although normally there is only one character in a read table having each of the list- and string-delimiting syntax classes (such as LEFTPAREN), it is perfectly acceptable for any character to have any syntax class, and for more than one to have the same class.

Note that a "syntax class" is an abstraction: there is no object referencing a collection of characters called a *syntax class.* Instead, a read table provides the *association* between a character and its syntax class, and the input/output routines enforce the abstraction by using read tables to drive the parsing.

The functions below are used to obtain and set the syntax class of a character in a read table. *CH* can either be a character code (a integer), or a character (a single-character atom). Single-digit integers are interpreted as character codes, rather than as characters. For example, 1 indicates Control-A, and 49 indicates the *character* 1. Note that *CH* can be a full sixteen-bit NS character (see Chapter 2).

> Note: Terminal tables, described in Chapter 30, also associate characters with syntax classes, and they can also be manipulated with the functions below. The set of read table and terminal table syntax classes are disjoint, so there is never any ambiguity about which type of table is being referred to.

(**GETSYNTAX** *CH TABLE*)   [Function]

> Returns the syntax class of *CH*, a character or a character code, with respect to *TABLE*. *TABLE* can be NIL, T, ORIG, or a real read table or terminal table.

CH can also be a syntax class, in which case GETSYNTAX returns a list of the character codes in *TABLE* that have that syntax class.

(**SETSYNTAX** *CHAR CLASS TABLE*)   [Function]

Sets the syntax class of *CHAR*, a character or character code, in *TABLE*. *TABLE* can be either NIL, T, or a real read table or terminal table. SETSYNTAX returns the previous syntax class of *CHAR*. *CLASS* can be any one of the following:

- The name of one of the basic syntax classes.

- A list, which is interpreted as a read macro (see below).

- NIL, T, ORIG, or a real read table or terminal table, which means to give *CHAR* the syntax class it has in the table indicated by *CLASS*. For example, (SETSYNTAX '%('ORIG TABLE) gives the left parenthesis character in *TABLE* the same syntax class that it has in the original system read table.

- A character code or character, which means to give *CHAR* the same syntax class as the character *CHAR* in *TABLE*. For example, (SETSYNTAX '{ '%[TABLE) gives the left brace character the same syntax class as the left bracket.

(**SYNTAXP** *CODE CLASS TABLE*)   [Function]

CODE is a character code; *TABLE* is NIL, T, or a real read table or terminal table. Returns T if *CODE* has the syntax class *CLASS* in *TABLE*; NIL otherwise.

CLASS can also be a read macro type (MACRO, SPLICE, INFIX), or a read macro option (FIRST, IMMEDIATE, etc.), in which case SYNTAXP returns T if the syntax class is a read macro with the specified property.

SYNTAXP will *not* accept a character as an argument, only a character *code*.

For convenience in use with SYNTAXP, the atom BREAK may be used to refer to *all* break characters, i.e., it is the union of LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, and BREAKCHAR. For purely symmetrical reasons, the atom SEPR corresponds to all separator characters. However, since the only separator characters are those that also appear in SEPRCHAR, SEPR and SEPRCHAR are equivalent.

Note that GETSYNTAX never returns BREAK or SEPR as a value although SETSYNTAX and SYNTAXP accept them as arguments. Instead, GETSYNTAX returns one of the disjoint basic syntax classes that comprise BREAK. BREAK as an argument to SETSYNTAX is interpreted to mean BREAKCHAR if the character is not already of one of the BREAK classes. Thus, if %(is of class LEFTPAREN, then (SETSYNTAX '%('BREAK) doesn't do anything, since %(is already a break character, but (SETSYNTAX '%('BREAKCHAR) means make %(be *just* a break character, and therefore disables the LEFTPAREN function of %(. Similarly, if one of the format characters is disabled completely, e.g., by (SETSYNTAX '%('OTHER), then (SETSYNTAX '%('BREAK) would make %(be *only* a break character; it would *not* restore %(as LEFTPAREN.

The following functions provide a way of collectively accessing and setting the separator and break characters in a read table:

(**GETSEPR** *RDTBL*)   [Function]

Returns a list of separator character codes in *RDTBL*. Equivalent to (GETSYNTAX 'SEPR *RDTBL*).

(**GETBRK** *RDTBL*)   [Function]

Returns a list of break character codes in *RDTBL*. Equivalent to (GETSYNTAX 'BREAK *RDTBL***)**.

(**SETSEPR** *LST FLG RDTBL*)   [Function]

Sets or removes the separator characters for *RDTBL*. *LST* is a list of charactors or character codes. *FLG* determines the action of SETSEPR as follows: If *FLG*=NIL, makes *RDTBL* have exactly the elements of *LST* as separators, discarding from *RDTBL* any old separator characters not in *LST*. If *FLG*=0, removes from *RDTBL* as separator characters all elements of *LST*. This provides an "UNSETSEPR". If *FLG*=1, makes each of the characters in *LST* be a separator in *RDTBL*.

If *LST*=T, the separator characters are reset to be those in the system's read table for terminals, regardless of the value of *FLG*, i.e., (SETSEPR T) is equivalent to (SETSEPR (GETSEPR T)). If *RDTBL* is T, then the characters are reset to those in the original system table.

Returns NIL.

(**SETBRK** *LST FLG RDTBL*)   [Function]

Sets the break characters for *RDTBL*. Similar to SETSEPR.

As with SETSYNTAX to the BREAK class, if any of the list- or string-delimiting break characters are disabled by an appropriate SETBRK (or by making it be a separator character), its special action for READ will *not* be restored by simply making it be a break character again with SETBRK. However, making these characters be break characters when they already *are* will have no effect.

The action of the ESCAPE character (normally %) is not affected by SETSEPR or SETBRK. It can be disabled by setting its syntax to the class OTHER, and other characters can be used for escape on input by assigning them the class ESCAPE. As of this writing, however, there is no way to change the output escape character; it is "hardwired" as %. That is, on output, characters of special syntax that need to be preceded by the ESCAPE character will always be preceded by %, independent of the syntax of % or which, if any characters, have syntax ESCAPE.

The following function can be used for defeating the action of the ESCAPE character or characters:

(**ESCAPE** *FLG RDTBL*) [Function]

> If *FLG*=NIL, makes characters of class ESCAPE behave like characters of class OTHER on input. Normal setting is (ESCAPE T). ESCAPE returns the previous setting.

Read Macros

This is a description of the OLD-INTERLISP-T read macros. Read macros are user-defined syntax classes that can cause complex operations when certain characters are read. Read macro characters are defined by specifying as a syntax class an expression of the form:

$$(TYPE\ OPTION_1\ ...\ OPTION_N\ FN)$$

where *TYPE* is one of MACRO, SPLICE, or INFIX, and *FN* is the name of a function or a lambda expression. Whenever READ encounters a read macro character, it calls the associated function, giving it as arguments the input stream and read table being used for that call to READ. The interpretation of the value returned depends on the type of read macro:

MACRO   This is the simplest type of read macro. The result returned from the macro is treated as the expression to be read, instead of the read macro character. Often the macro reads more input itself. For example, in order to cause ~EXPR to be read as (NOT EXPR), one could define ~ as the read macro:

```
[MACRO (LAMBDA (FL RDTBL)
               (LIST 'NOT (READ FL RDTBL]
```

SPLICE   The result (which should be a list or NIL) is spliced into the input using NCONC. For example, if $ is defined by the read macro:

```
(SPLICE (LAMBDA NIL (APPEND FOO)))
```

and the value of FOO is (A B C), then when you input (X $ Y), the result will be (X A B C Y).

INFIX   The associated function is called with a third argument, which is a list, in TCONC format (Chapter 3), of what has been read at the current level of list nesting. The function's value is taken as a new TCONC list which replaces the old one. For example, the infix operator + could be defined by the read macro:

```
(INFIX (LAMBDA (FL RDTBL Z)
               (RPLACA (CDR Z)
                   (LIST (QUOTE IPLUS)
                         (CADR Z)
                         (READ FL RDTBL)))
               Z))
```

If an INFIX read macro character is encountered *not* in a list, the third argument to its associated function is NIL. If the function returns NIL, the read macro character is essentially ignored and reading continues. Otherwise, if the function returns a TCONC list of one element, that element is the value of

the `READ`. If it returns a `TCONC` list of more than one element, the list is the value of the `READ`.

The specification for a read macro character can be augmented to specify various options $OPTION_1$ ... $OPTION_N$, e.g., (`MACRO FIRST IMMEDIATE` *FN*). The following three disjoint options specify when the read macro character is to be effective:

ALWAYS The default. The read macro character is always effective (except when preceded by the `%` character), and is a break character, i.e., a member of (`GETSYNTAX 'BREAK` *RDTBL*).

FIRST The character is interpreted as a read macro character *only* when it is the first character seen after a break or separator character; in all other situations, the character is treated as having class `OTHER`. The read macro character is *not* a break character. For example, the quote character is a `FIRST` read macro character, so that `DON'T` is read as the single atom `DON'T`, rather than as `DON` followed by (`QUOTE T`).

ALONE The read macro character is *not* a break character, and is interpreted as a read macro character only when the character would have been read as a separate atom if it were not a read macro character, i.e., when its immediate neighbors are both break or separator characters.

Making a `FIRST` or `ALONE` read macro character be a break character (with `SETBRK`) disables the read macro interpretation, i.e., converts it to syntax class `BREAKCHAR`. Making an `ALWAYS` read macro character be a break character is a no-op.

The following two disjoint options control whether the read macro character is to be protected by the `ESCAPE` character on output when a litatom containing the character is printed:

ESCQUOTE or ESC The default. When printed with `PRIN2`, the read macro character will be preceded by the output escape character (`%`) as needed to permit the atom containing it to be read correctly. Note that for `FIRST` macros, this means that the character need be quoted only when it is the first character of the atom.

NOESCQUOTE or NOESC The read macro character will always be printed without an escape. For example, the **?** read macro in the `T` read table is a `NOESCQUOTE` character. Unless you are very careful what you are doing, read macro characters in `FILERDTBL` should never be `NOESCQUOTE`, since symbols that happen to contain the read macro character will not read back in correctly.

The following two disjoint options control when the macro's function is actually executed:

IMMEDIATE or IMMED The read macro character is immediately activated, i.e., the current line is terminated, as if an `EOL` had been typed, a carriage-return line-feed is printed,

and the entire line (including the macro character) is passed to the input function.

IMMEDIATE read macro characters enable you to specify a character that will take effect immediately, as soon as it is encountered in the input, rather than waiting for the line to be terminated. Note that this is not necessarily as soon as the character is *typed*. Characters that cause action as soon as they are typed are interrupt characters (see Chapter 30).

Note that since an IMMEDIATE macro causes any input before it to be sent to the reader, characters typed before an IMMEDIATE read macro character cannot be erased by Control-A or Control-Q once the IMMEDIATE character has been typed, since they have already passed through the line buffer. However, an INFIX read macro can still alter some of what has been typed earlier, via its third argument.

NONIMMEDIATE or NONIMMED The default. The read macro character is a normal character with respect to the line buffering, and so will not be activated until a carriage-return or matching right parenthesis or bracket is seen.

Making a read macro character be both ALONE and IMMEDIATE is a contradiction, since ALONE requires that the next character be input in order to see if it is a break or separator character. Thus, ALONE read macros are always NONIMMEDIATE, regardless of whether or not IMMEDIATE is specified.

Read macro characters can be "nested". For example, if = is defined by

```
(MACRO (LAMBDA (FL RDTBL)
    (EVAL (READ FL RDTBL))))
```

and ! is defined by

```
(SPLICE (LAMBDA (FL RDTBL)
    (READ FL RDTBL)))
```

then if the value of FOO is (A B C), and (X =FOO Y) is input, (X (A B C) Y) will be returned. If (X !=FOO Y) is input, (X A B C Y) will be returned.

Note: If a read macro's function calls READ, and the READ returns NIL, the function cannot distinguish the case where a RIGHTPAREN or RIGHTBRACKET followed the read macro character, (e.g. "(A B ')"), from the case where the atom NIL (or "()") actually appeared. In Interlisp-D, a READ inside of a read macro when the next input character is a RIGHTPAREN or RIGHTBRACKET reads the character and returns NIL, just as if the READ had not occurred inside a read macro.

If a call to READ from within a read macro encounters an unmatched RIGHTBRACKET *within* a list, the bracket is simply put back into the buffer to be read (again) at the higher level. Thus, inputting an expression such as (A B '(C D] works correctly.

(**INREADMACROP**) [Function]

> Returns NIL if currently *not* under a read macro function, otherwise the number of unmatched left parentheses or brackets.

(**READMACROS** *FLG RDTBL*) [Function]

> If *FLG*=NIL, turns off action of read macros in read table *RDTBL*. If *FLG*=T, turns them on. Returns previous setting.
>
> The following read macros are standardly defined in Interlisp in the T and EDITRDTBL read tables:

' (single-quote)
> Returns the next expression, wrapped in a call to QUOTE; e.g., 'FOO reads as (QUOTE FOO). The macro is defined as a FIRST read macro, so that the quote character has no effect in the middle of a symbol. The macro is also ignored if the quote character is immediately followed by a separator character.

Control-Y
> Defined in T and EDITRDTBL. Returns the result of evaluating the next expression. For example, if the value of FOO is (A B), then (LIST 1 *control-Y*FOO 2) is read as (LIST 1 (A B) 2). Note that no structure is copied; the third element of that input expression is still EQ to the value of FOO. Control-Y can thus be used to read structures that ordinarily have no read syntax. For example, the value returned from reading (KEY1 *Control-Y*(ARRAY 10)) has an array as its second element. Control-Y can be thought of as an "un-quote" character. The choice of character to perform this function is changeable with SETTERMCHARS (see Chapter 16).

` (backquote)
> Backquote makes it easier to write programs to construct complex data structures. Backquote is like quote, except that within the backquoted expression, forms can be evaluated. The general idea is that the backquoted expression is a "template" containing some constant parts (as with a quoted form) and some parts to be filled in by evaluating something. Unlike with control-Y, however, the evaluation occurs not at the time the form is read, but at the time the backquoted expression is evaluated. That is, the backquote macro returns an expression which, when evaluated, produces the desired structure.
>
> Within the backquoted expression, the character "**,**" (comma) introduces a form to be evaluated. The value of a form preceded by ",@" is to be spliced in, using APPEND. If it is permissible to destroy the list being spliced in (i.e., NCONC may be used in the translation), then ",." can be used instead of ",@".
>
> For example, if the value of FOO is (1 2 3 4), then the form

```
'(A ,(CAR FOO) ,@(CDDR FOO) D E)
```

evaluates to `(A 1 3 4 D E)`; it is logically equivalent to writing

```
(CONS 'A
   (CONS (CAR FOO)
      (APPEND (CDDR FOO) '(D E))))
```
.

Backquote is particularly useful for writing macros. For example, the body of a macro that refers to X as the macro's argument list might be

```
'(COND
   ((FIXP ,(CAR X))
      ,(CADR X))
   (T .,(CDDR X)))
```

which is equivalent to writing

```
(LIST 'COND
    (LIST (LIST 'FIXP (CAR X))
          (CADR X))
    (CONS 'T (CDDR X)))
```

Note that comma does *not* have any special meaning outside of a backquote context.

For users without a backquote character on their keyboards, backquote can also be written as `|'` (vertical-bar, quote).

?   Implements the `?=` command for on-line help regarding the function currently being "called" in the typein (see Chapter 26).

| (vertical bar)   When followed by an end of line, tab or space, `|` is ignored, i.e., treated as a separator character, enabling the editor's CHANGECHAR feature (see Chapter 26). Otherwise it is a "dispatching" read macro whose meaning depends on the character(s) following it. The following are currently defined:

' (quote) -- A synonym for backquote.

. (period) -- Returns the evaluation of the next expression, i.e., this is a synonym for Control-Y.

, (comma) -- Returns the evaluation of the next expression *at load time*, i.e., the following expression is quoted in such a manner that the compiler treats it as a literal whose value is not determined until the compiled expression is loaded.

O or o (the letter O) -- Treats the next number as octal, i.e., reads it in radix 8. For example, `|o12 = 10` (decimal).

B or b -- Treats the next number as binary, i.e., reads it in radix 2. For example, `|b101 = 5` (decimal).

`X` or `x` -- Treats the next number as hexadecimal, i.e., reads it in radix 16. The uppercase letters `A` though `F` are used as the digits after `9`. For example, `|x1A` `= 26` (decimal).

`R` or `r` -- Reads the next number in the radix specified by the (decimal) number that appears between the `|` and the `R`. When inputting a number in a radix above ten, the upper-case letters `A` through `Z` can be used as the digits after `9` (but there is no digit above `Z`, so it is not possible to type all base-99 digits). For example, `|3r120` reads 120 in radix 3, returning 15.

`(`, `{`, `^` -- Used internally by `HPRINT` and `HREAD` (see above) to print and read unusual expressions.

The dispatching characters that are letters can appear in either upper- or lowercase.

25.   USER/ INPUT/OUTPUT PACKAGES

Interlisp-D can perform input/output operations on a large variety of physical devices.

This chapter presents a number of packages that have been developed for displaying and allowing the user to enter information. These packages are used to implement the user interface of many system facilities.

> INSPECT (see the INSPECT section below) provides a window-based facility for displaying and changing the fields of a data object.

> PROMPTFORWORD (see the PROMPTFORWORD section below) is a function used for entering a simple string of characters. Basic editing and prompting facilities are provided.

> ASKUSER (see the ASKUSER section below) provides a more complicated prompting and answering facility, allowing a series of questions to be printed. Prompts and argument completion are supported.

> TTYIN (see the TTYIN Display Typein Editor section below) is a display typein editor, that provides complex text editing facilities when entering an input line.

> PRETTYPRINT (see the Prettyprint section below) is used for printing function definitions and other list structures, using multiple fonts and indenting lines to show the structure of the list.

Inspector

The Inspector provides a display-oriented facility for looking at and changing arbitrary Interlisp-D data structures. The inspector can be used to inspect all user datatypes and many system datatypes (although some objects such as numbers have no inspectable structure). The inspector displays the field names and values of an arbitrary object in a window that allows setting of the properties and further inspection of the values. This latter feature makes it possible to "walk" around all of the data structures in the system at the touch of a button. In addition, the inspector is integrated with the break package to allow inspection of any object on the stack and with the display and teletype structural editors to allow the editors to be used to "inspect" list structures and the inspector to "edit" datatypes.

The underlying mechanisms of the data inspector have been designed to allow their use as specialized editors in user applications. This functionality is described at the end of this section.

> Note: Currently, the inspector does not have UNDOing. Also, variables whose values are changed will not be marked as such.

Calling the Inspector

There are several ways to open an inspect window onto an object. In addition to calling INSPECT directly (below), the inspector can also be called by buttoning an Inspect command inside an existing inspector window. Finally, if a non-list is edited with EDITDEF (see Chapter 17), the inspector is called. This also causes the inspector to be called by the Dedit command from the display editor or the EV command from the teletype editor if the selected piece of structure is a non-list.

(**INSPECT** *OBJECT ASTYPE WHERE*)   [Function]

> Creates an inspect window onto *OBJECT*. If *ASTYPE* is given, it will be taken as the record type of *OBJECT*. This allows records to be inspected with their property names. If *ASTYPE* is NIL, the data type of *OBJECT* will be used to determine its property names in the inspect window.

> *WHERE* specifies the location of the inspect window. If *WHERE* is NIL, the user will be prompted for a location. If *WHERE* is a window, it will be used as the inspect window. If *WHERE* is a region, the inspect window will be created in that region the screen. If *WHERE* is a position, the inspect window will have its lower left corner at that position on the screen.

> INSPECT returns the inspect window onto *OBJECT*, or NIL if no inspection took place.

(**INSPECTCODE** *FN WHERE* — — — —)   [Function]

> Opens a window and displays the compiled code of the function *FN* using PRINTCODE. The window is scrollable.

> *WHERE* determines where the window should appear. It can be a position, a region, or a window. If NIL, the user is prompted to specify the position of the window.

> Note: If the Tedit library package is loaded, INSPECTCODE uses it to create the code inspector window. Also, if INSPECTCODE is called to inspect the frame name in a break window (see Chapter 14), the location in the code that the frame's PC indicates it was executing at the time is highlighted.

Multiple Ways of Inspecting

For some datatypes there is more than one aspect that is of interest or more than one method of inspecting the object. In these cases, the inspector will bring up a menu of the possibilities and wait for the user to select one.

If the object is a litatom, the commands are the types for which the litatom has definitions as determined by HASDEF. Some typical commands are:

> FNS   Edit the definition of the selected litatom.

> VARS   Inspect the value.

> PROPS Inspect the property list.

> If the object is a list, there will be choice of how to inspect the list:

`Inspect`	Opens an inspect window in which the properties are numbers and the values are the elements of the list.
`TtyEdit`	Calls the teletype list structure editor on the list (see Chapter 16).
`DisplayEdit`	Calls the DEdit display editor on the list (see Chapter 16).
As a `PLIST`	Inspects the list as a property list, if the list is in property list form: $(\text{PROP}_1\ \text{VAL}_1)\ \ldots\ (\text{PROP}_N\ \text{VAL}_N))$.
As an `ALIST`	Inspects the list as an association-list, if the list is in `ASSOC` list form: $(\text{PROP}_1\ \text{VAL}_1\ \ldots\ \text{PROP}_N\ \text{VAL}_N)$.
As a record	Brings up a submenu with all of the `RECORD`s in the system and inspect the list with the one chosen.
As a "record type"	Inspects the list as the record of the type named in its `CAR`, if the `CAR` of the list is the name of a `TYPERECORD` (see Chapter 8).

If the object is a bitmap, the choice is between inspecting the bitmap's contents with the bitmap editor (`EDITBM`) or inspecting the bitmap's fields.

Other datatypes may include multiple methods for inspecting objects of that type.

Inspect Windows

An inspect window displays two columns of values. The lefthand column lists the property names of the structure being inspected. The righthand column contains the values of the properties named on the left. For variable length data such as lists and arrays, the "property names" are numbers from 1 to the length of the inspected item and the values are the corresponding elements. For arrays, the property names are the array element numbers and the values are the corresponding elements of the array.

For large lists or arrays, or datatypes with many fields, the initial window may be too small to contain all of them. In these cases, the unseen elements can be scrolled into view (from the bottom) or the window can be reshaped to increase its size.

In an inspect window, the LEFT button is used to select things, the MIDDLE button to invoke commands that apply to the selected item. Any property or value can be selected by pointing the cursor directly at the text representing it, and clicking the LEFT button. There is one selected item per window and it is marked by having its surrounding box inverted.

The options offered by the MIDDLE button depend on whether the selection is a property or a value. If the selected item is a value, the options provide different ways of inspecting the selected structure. The exact commands that are given depend on the type of the value. An example of the menu you may see is:

```
DisplayEdit
  TtyEdit
  Inspect
As a record
As a PLIST
```

If the selected item is a property name, the command SET will appear. If selected, the user will be asked to type in an expression, and the selected property will be set to the result of evaluating the read form. The evaluation of the read form and the replacement of the selected item property will appear as their own history events and are individually undoable. Properties of system datatypes cannot be set. (There are often consistency requirements which can be inadvertently violated in ways that crash the system. This may be true of some user datatypes as well, however the system doesn't know which ones. Users are advised to exercise caution.)

It is possible to copy-select property names or values out of an inspect window. Litatoms, numbers and strings are copied as they are displayed. Unprintable objects (such as bitmaps, etc.) come out as an appropriate system expression, such that if is evaluated, the object is re-created.

Inspect Window Commands

By pressing the MIDDLE button in the title of the inspect window, a menu of commands that apply to the inspect window is brought up:

```
ReFetch
IT←datum
IT←selection
```

ReFetch   [Inspect Window Command]

> An inspect window is not automatically updated when the structure it is inspecting is changed. The ReFetch command will refetch and redisplay all of the fields of the object being inspected in the inspect window.

IT←datum   [Inspect Window Command]

> Sets the variable IT to object being inspected in the inspect window.

IT←selection   [Inspect Window Command]

> Sets the variable IT to the property name or value currently selected in the inspect window.

Interaction With Break Windows

The break window facility (see Chapter 14) knows about the inspector in the sense that the backtrace frame window is an inspect window onto the frame selected from the back trace menu during a break. Thus you can call the inspector on an object that is bound on the stack by selecting its frame in the back trace menu, selecting its value with the LEFT button in the back trace frame window, and

selecting the inspect command with the `MIDDLE` button in the back trace frame window. The values of variables in frames can be set by selecting the variable name with the `LEFT` button and then the "Set" command with the `MIDDLE` button.

> Note: The inspector will only allow the setting of named variables. Even with this restriction it is still possible to crash the system by setting variables inside system frames. Exercise caution in setting variables in other than your own code.

Controlling the Amount Displayed During Inspection

The amount of information displayed during inspection can be controlled using the following variables:

MAXINSPECTCDRLEVEL   [Variable]

> The inspector prints only the first `MAXINSPECTCDRLEVEL` elements of a long list, and will make the tail containing the unprinted elements the last item. The last item can be inspected to see further elements. Initially 50.

MAXINSPECTARRAYLEVEL   [Variable]

> The inspector prints only the first `MAXINSPECTARRAYLEVEL` elements of an array. The remaining elements can be inspected by calling the function (`INSPECT/ARRAY ARRAY BEGINOFFSET`) which inspects the `BEGINOFFSET` through the `BEGINOFFSET` + `MAXINSPECTARRAYLEVEL` elements of `ARRAY`. Initially 300.

INSPECTPRINTLEVEL   [Variable]

> When printing the values, the inspector resets `PRINTLEVEL` (see Chapter 25) to the value of `INSPECTPRINTLEVEL`. Initially (2 . 5).

INSPECTALLFIELDSFLG   [Variable]

> If `INSPECTALLFIELDSFLG` is `T`, the inspector will show computed fields (`ACCESSFNS`, Chapter 8) as well as regular fields for structures that have a record definition. Initially `T`.

Inspect Macros

The Inspector can be extended to inspect new structures and datatypes by adding entries to the list `INSPECTMACROS`. An entry should be of the form (`OBJECTTYPE . INSPECTINFO`). `OBJECTTYPE` is used to determine the types of objects that are inspected with this macro. If `OBJECTTYPE` is a litatom, the `INSPECTINFO` will be used to inspect items whose type name is `OBJECTTYPE`. If `OBJECTTYPE` is a list of the form (`FUNCTION DATUM-PREDICATE`), `DATUM-PREDICATE` will be `APPLY`ed to the item and if it returns non-`NIL`, the `INSPECTINFO` will be used to inspect the item.

`INSPECTINFO` can be one of two forms. If `INSPECTINFO` is a litatom, it should be a function that will be applied to three arguments (the item being inspected, `OBJECTTYPE`, and the value of `WHERE` passed to `INSPECT`) that should do the inspection. If `INSPECTINFO` is not a litatom, it should be a list

of (PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN) where the elements of this list are the arguments for INSPECTW.CREATE, described below. From this list, the WHERE argument will be evaluated; the others will not. If WHERE is NIL, the value of WHERE that was passed to INSPECT will be used.

Examples:

The entry ((FUNCTION MYATOMP) PROPNAMES GETPROP PUTPROP) on INSPECTMACROS would cause all objects satisfying the predicate MYATOMP to have their properties inspected with GETPROP and PUTPROP. In this example, MYATOMP should make sure the object is a litatom.

The entry (MYDATATYPE . MYINSPECTFN) on INSPECTMACROS would cause all datatypes of type MYDATATYPE to be passed to the function MYINSPECTFN.

INSPECTWs

The inspector is built on the abstraction of an INSPECTW. An INSPECTW is a window with certain window properties that display an object and respond to selections of the object's parts. It is characterized by an object and its list of properties. An INSPECTW displays the object in two columns with the property names on the left and the values of those properties on the right. An INSPECTW supports the protocol that the LEFT mouse button can be used to select any property name or property value and the MIDDLE button calls a user provided function on the selected value or property. For the Inspector application, this function puts up a menu of the alternative ways of inspecting values or of the ways of setting properties. INSPECTWs are created with the following function:

> (**INSPECTW.CREATE** *DATUM PROPERTIES FETCHFN STOREFN PROPCOMMANDFN VALUECOMMANDFN TITLECOMMANDFN TITLE SELECTIONFN WHERE PROPPRINTFN*)
> [Function]

> > Creates an INSPECTW that views the object *DATUM*. If *PROPERTIES* is a list, it is taken as the list of properties of *DATUM* to display. If *PROPERTIES* is a litatom, it is APPLYed to *DATUM* and the result is used as the list of properties to display.

> > *FETCHFN* is a function of two arguments (OBJECT PROPERTY) that should return the value of the PROPERTY property of OBJECT. The result of this function will be printed (with PRIN2) in the INSPECTW as the value.

> > *STOREFN* is a function of three arguments (OBJECT PROPERTY NEWVALUE) that changes the PROPERTY property of OBJECT to NEWVALUE. It is used by the default *PROPCOMMANDFN* and *VALUECOMMANDFN* to change the value of a property and also by the function INSPECTW.REPLACE (described below). This can be NIL if the user provides command functions which do not call INSPECTW.REPLACE. Each replace action will be a separate event on the history list. Users are encouraged to provide UNDOable STOREFNs.

> > *PROPCOMMANDFN* is a function of three arguments (PROPERTY OBJECT INSPECTW) which gets called when the user presses the MIDDLE button and the selected item in the

`INSPECTW` is a property name. `PROPERTY` will be the name of the selected property, `OBJECT` will be the datum being viewed, and `INSPECTW` will be the window. If *PROPCOMMANDFN* is a string, it will get printed in the `PROMPTWINDOW` when the `MIDDLE` button is pressed. This provides a convenient way to notify the user about disabled commands on the properties. `DEFAULT.INSPECTW.PROPCOMMANDFN`, the default *PROPCOMMANDFN*, will present a menu with the single command `Set` on it. If selected, the `Set` command will read a value from the user and set the selected property to the result of `EVAL`uating this read value.

VALUECOMMANDFN is a function of four arguments (`VALUE    PROPERTY    OBJECT    INSPECTW`) that gets called when the user presses the `MIDDLE` button and the selected item in the `INSPECTW` is a property value. `VALUE` will be the selected value (as returned by *FETCHFN*), `PROPERTY` will be the name of the property `VALUE` is the value of, `OBJECT` will be the datum being viewed, and `INSPECTW` will be the `INSPECTW` window.

`DEFAULT.INSPECTW.VALUECOMMANDFN`, the default *VALUECOMMANDFN*, will present a menu of possible ways of inspecting the value and create a new Inspect window if one of the menu items is selected.

TITLECOMMANDFN is a function of two arguments (`INSPECTW    OBJECT`) which gets called when the user presses the `MIDDLE` button and the cursor is in the title or border of the inspect window `INSPECTW`. This command function is provided so that users can implement commands that apply to the entire object. The default *TITLECOMMANDFN* (`DEFAULT.INSPECTW.TITLECOMMANDFN`) presents a menu with the commands `ReFetch`, IT←datum, and IT←election .

TITLE specifies the title of the window. If *TITLE* is `NIL`, the title of the window will be the printed form of *DATUM* followed by the string " Inspector". If *TITLE* is the litatom `DON'T`, the inspect window will not have a title. If *TITLE* is any other litatom, it will be applied to the *DATUM* and the potential inspect window (if it is known). If this result is the litatom `DON'T`, the inspect window will not have a title; otherwise the result will be used as a title. If *TITLE* is not a litatom, it will be used as the title.

SELECTIONFN is a function of three arguments (`PROPERTY    VALUEFLG    INSPECTW`) which gets called when the user releases the left button and the cursor is on one of the items. The *SELECTIONFN* allows a program to take action on the user's selection of an item in the inspect window. At the time this function is called, the selected item has been "selected". The function `INSPECTW.SELECTITEM` (described below) can be used to turn off this selection. `PROPERTY` will be the name of the property of the selected item. *VALUEFLG* will be `NIL` if the selected item is the property name; `T` if the selected item is the property value.

WHERE indicates where the inspect window should go. Its interpretation is described in `INSPECT` (see above).

PROPPRINTFN is a function of two arguments (`PROPERTY    DATUM`) which gets called to determine what to print in the property place for the property `PROPERTY`. If *PROPPRINTFN* returns `NIL`, no property name will be printed and the value will be printed to the left of the other values.

An inspect window uses the following window property names to hold information: *DATUM*, *FETCHFN*, *STOREFN*, *PROPCOMMANDFN*, *VALUECOMMANDFN*, *SELECTIONFN*, *PROPPRINTFN*, *INSPECTWTITLE*, *PROPERTIES*, *CURRENTITEM* and *SELECTABLEITEMS*.

(**INSPECTW.REDISPLAY** *INSPECTW PROPS* —)   [Function]

Updates the display of the objects being inspected in INSPECTW. If *PROPS* is a property name or a list of property names, only those properties are updated. If *PROPS* is NIL, all properties are redisplayed. This function is provided because inspect windows do not automatically update their display when the object they are showing changes.

This function is called by the ReFetch command in the title command menu of an INSPECTW (see above).

(**INSPECTW.REPLACE** *INSPECTW PROPERTY NEWVALUE*)   [Function]

Calls the STOREFN of the inspect window *INSPECTW* to change the property named *PROPERTY* to the value *NEWVALUE* and updates the display of *PROPERTY*'s value in the display. This provides a functional interface for user PROPCOMMANDFNs.

(**INSPECTW.SELECTITEM** *INSPECTW PROPERTY VALUEFLG*)   [Function]

Sets the selected item in an inspect window. The item is inverted on the display and put on the window property CURRENTITEM of *INSPECTW*. If *INSPECTW* has a CURRENTITEM, it is deselected. *PROPERTY* is the name of the property of the selected item. *VALUEFLG* is NIL if the selected item is the property name; T if the selected item is the property value. If *PROPERTY* is NIL, no item will be selected. This provides a way of deselecting all items.

PROMPTFORWORD

PROMPTFORWORD is a function that reads in a sequence of characters, generally from the keyboard, without involving READ-like syntax. A user can supply a prompting string, as well as a "candidate" string, which is printed and used if the user types only a word terminator character (or doesn't type anything before a given time limit). As soon as any characters are typed the "candidate" string is erased and the new input takes its place.

PROMPTFORWORD accepts user type-in until one of the "word terminator" characters is typed. Normally, the word terminator characters are EOL, ESCAPE, LF, SPACE, or TAB. This list can be changed using the TERMINCHAR.LST argument to PROMPTFORWORD, for example if it is desirable to allow the user to input lines including spaces.

PROMPTFORWORD also recognizes the following special characters:

Control-A
BACKSPACE
    DELETE   Any of these characters deletes the last character typed and appropriately erases it from the echo stream if it is a display stream.

Control-Q	Erases all the type-in so far.
Control-R	Reprints the accumulated string.
Control-V	"Quotes" the next character: after typing Control-V, the next character typed is added to the accumulated string, regardless of any special meaning it has. Allows the user to include editing characters and word terminator characters in the accumulated string.
Control-W	Erases the last word.
?	Calls up a "help" facility. The action taken is defined by the GENERATE?LIST.FN argument to PROMPTFORWORD (see below). Normally, this prints a list of possible candidates.

(**PROMPTFORWORD** *PROMPT.STR CANDIDATE.STR GENERATE?LIST.FN ECHO.CHANNEL DONTECHOTYPEIN.FLG URGENCY.OPTION TERMINCHARS.LST KEYBD.CHANNEL*)
[Function]

PROMPTFORWORD has a multiplicity of features, which are specified through a rather large number of input arguments, but the default settings for them (i.e., when they aren't given, or are given as NIL) is such to minimize the number needed in the average case, and an attempt has been made to order the more frequently non-defaulted arguments at the beginning of the argument list. The default input and echo are both to the terminal; the terminal table in effect during input allows most control characters to be INDICATE'd.

PROMPTFORWORD returns NIL if a null string is typed; this would occur when no candidate is given and only a terminator is typed, or when the candidate is erased and a terminator is typed with no other input still un-erased. In all other cases, PROMPTFORWORD returns a string.

PROMPTFORWORD is controlled through the following arguments:

PROMPT.STR	If non-NIL, this is coerced to a string and used for prompting; an additional space is output after this string.
CANDIDATE.STR	If non-NIL, this is coerced to a string and offered as initial contents of the input buffer.
GENERATE?LIST.FN	If non-NIL, this is either a string to be printed out for help, or a function to be applied to *PROMPT.STR* and *CANDIDATE.STR* (after both have been coerced to strings), and which should return a list of potential candidates. The help string or list of potential candidates will then be printed on a separate line, the prompt will be restarted, and any type-in will be re-echoed. Note: If *GENERATE?LIST.FN* is a function, its value list will be cached so that it will be run at most once per call to PROMPTFORWORD.
ECHO.CHANNEL	Coerced to an output stream; NIL defaults to T, the "terminal output stream", normally (TTYDISPLAYSTREAM). To achieve echoing to the "current output stream", use (GETSTREAM NIL 'OUTPUT). If echo is to a display stream, it will have a flashing caret showing where the next input is to be echoed.
DONTECHOTYPEIN.FLG	If T, there is no echoing of the input characters. If the value of *DONTECHOTYPEIN.FLG* is a single-character atom or string, that character is

	echoed instead of the actual input. For example, LOGIN prompts for a password with *DONTECHOTYPEIN.FLG* being "*".
URGENCY.OPTION	If NIL, PROMPTFORWORD quietly wait for input, as READ does; if a number, this is the number of seconds to wait for the user to respond (if timeout is reached, then *CANDIDATE.WORD* is returned, regardless of any other type-in activity); if T, this means to wait forever, but periodically flash the window to alert the user; if TTY, then PROMPTFORWORD grabs the TTY immediately. When URGENCY.OPTION = TTY, the cursor is temporarily changed to a different shape to indicate the urgent nature of the request.
TERMINCHARS.LST	This is list of "word terminator" character codes; it defaults to (CHARCODE (EOL ESCAPE LF SPACE TAB)). This may also be a single character code.
KEYBD.CHANNEL	If non-NIL, this is coerced to a stream, and the input bytes are taken from that stream. NIL **defaults to the** keyboard input stream. Note that this is not the same as the terminal input stream T, which is a buffered keyboard input stream, not suitable for use with PROMPTFORWORD.

Examples:

```
(PROMPTFORWORD
    "What is your FOO word?" 'Mumble
    (FUNCTION (LAMBDA () '(Grumble Bletch)))
    PROMPTWINDOW NIL 30)
```

This first prompts the user for input by printing the first argument as a prompt into PROMPTWINDOW; then the proffered default answer, Mumble, is printed out and the caret starts flashing just after it to indicate that the upcoming input will be echoed there. If the user fails to complete a word within 30 seconds, then the result will be the string Mumble.

```
(FRESHLINE T)
(LIST
    (PROMPTFORWORD
        (CONCAT "{" HOST "} Login:")
        (USERNAME NIL NIL T))
    (PROMPTFORWORD
        " (password)" NIL NIL NIL '*))
```

This first prompts in whatever window is currently (TTYDISPLAYSTREAM), and then takes in a username; the second call prompts with (password) and takes in another word (the password) without proffering a candidate, echoing the typed-in characters as "*".

ASKUSER

DWIM, the compiler, the editor, and many other system packages all use ASKUSER, an extremely general user interaction package, for their interactions with the user at the terminal. ASKUSER takes as its principal argument KEYLST which is used to drive the interaction. KEYLST specifies what the user can type at any given point, how ASKUSER should respond to the various inputs, what value should be returned by ASKUSER, and is also used to present the user at any given point with a list of the

possible responses. ASKUSER also takes other arguments which permit specifying a wait time, a default value, a message to be printed on entry, a flag indicating whether or not typeahead is to be permitted, a flag indicating whether the transaction is to be stored on the history list (see Chapter 13), a default set of options, and an (optional) input file/string.

(**ASKUSER** *WAIT DEFAULT MESS KEYLST TYPEAHEAD LISPXPRNTFLG OPTIONSLST FILE*) [Function]

> *WAIT* is either NIL or a number (of seconds). *DEFAULT* is a single character or a sequence (list) of characters to be used as the default inputs for the case when *WAIT* is not NIL and more than *WAIT* seconds elapse without any input. In this case, the character(s) from *DEFAULT* are processed exactly as though they had been typed, except that ASKUSER first types "...".

> *MESS* is the initial message to be printed by ASKUSER, if any, and can be a string, or a list. In the latter case, each element of the list is printed, separated by spaces, and terminated with a " ? ". *KEYLST* and *OPTIONSLST* are described. *TYPEAHEAD* is T if the user is permitted to typeahead a response to ASKUSER. NIL means any typeahead should be cleared and saved. *LISPXPRNTFLG* determines whether or not the interaction is to be recorded on the history list. *FILE* can be either NIL (in which case it defaults to the terminal input stream, T) or a stream.

> All input operations take place from *FILE* until an unacceptable input is encountered, i.e., one that does not conform to the protocol defined by *KEYLST*. At that point, *FILE* is set to T, *DEFAULT* is set to NIL, the input buffer is cleared, and a bell is rung. Unacceptable inputs are not echoed.

> The value of ASKUSER is the result of packing all the keys that were matched, unless the *RETURN* option is specified (see the Options section below).

(**MAKEKEYLST** *LST DEFAULTKEY LCASEFLG AUTOCOMPLETEFLG*) [Function]

> *LST* is a list of atoms or strings. MAKEKEYLST returns an ASKUSER KEYLST which will permit the user to specify one of the elements on *LST* by either typing enough characters to make the choice unambiguous, or else typing a number between 1 and N, where N is the length of *LST*.

> For example, if ASKUSER is called with KEYLST = (MAKEKEYLST '(CONNECT SUPPORT COMPILE)), then the user can type C-O-N, S, C-O-M, 1, 2, or 3 to indicate one of the three choices.

> If *LCASEFLG* = T, then echoing of upper case elements will be in lower case (but the value returned will still be one of the elements of LST). If *DEFAULTKEY* is non-NIL, it will be the last key on the KEYLST. Otherwise, a key which permits the user to indicate "No - none of the above" choices, in which case the value returned by ASKUSER will be NIL.

> *AUTOCOMPLETEFLG* is used as the value of the *AUTOCOMPLETEFLG* option of the resulting key list.

Format of KEYLST

KEYLST is a list of elements of the form (KEY PROMPTSTRING . OPTIONS), where KEY is an atom or a string (equivalent), PROMPTSTRING is an atom or a string, and OPTIONS a list of options in property list format. The options are explained below. If an option is specified in OPTIONS, the value of the option is the next element. Otherwise, if the option is specified in the OPTIONSLST argument to ASKUSER, its value is the next element on OPTIONSLST. Thus, OPTIONSLST can be used to provide default options for an entire KEYLST, rather than having to include the option at each level. If an option does not appear on either OPTIONS or OPTIONSLST, its value is NIL.

For convenience, an entry on KEYLST of the form (KEY . ATOM/STRING), can be used as an abbreviation for (KEY ATOM/STRING CONFIRMFLG T), and an entry of just the form KEY, i.e., a non-list, as an abbreviation for (KEY NIL CONFIRMFLG T).

As each character is read, it is matched against the currently active keys. A character matches a key if it is the same character as that in the corresponding position in the key, or, if the character is an alphabetic character, if the characters are the same without regard for upper/lower case differences, i.e. "A" matches "a" and vice versa (unless the NOCASEFLG option is T, see the Options section below). In other words, if two characters have already been input and matched, the third character is matched with each active key by comparing it with the third character of that key. If the character matches with one or more of the keys, the entries on KEYLST corresponding to the remaining keys are discarded. If the character does not match with any of the keys, the character is not echoed, and a bell is rung instead.

When a key is complete, PROMPTSTRING is printed (NIL is equivalent to "", the empty string, i.e., nothing will be printed). Then, if the value of the CONFIRMFLG option is T, ASKUSER waits for confirmation of the key by a carriage return or space. Otherwise, the key does not require confirmation.

Then, if the value of the KEYLST option is not NIL, its value becomes the new KEYLST, and the process recurses. Otherwise, the key is a "leaf," i.e., it terminates a particular path through the original, top-level KEYLST, and ASKUSER returns the result of packing all the keys that have been matched and completed along the way (unless the RETURN option is used to specify some other value, as described below).

For example, when ASKUSER is called with KEYLST = NIL, the following KEYLST is used as the default:

                    ((Y "escr") (N "ocr"))

This KEYLST specifies that if (as soon as) the user types Y (or y), ASKUSER echoes with Y, prompts with escr, and returns Y as its value. Similarly, if the user types N, ASKUSER echoes the N, prompts with ocr, and returns N. If the user types ?, ASKUSER prints:

            Yes

            No

to indicate his possible responses. All other inputs are unacceptable, and ASKUSER will ring the bell and not echo or print anything.

For a more complicated example, the following is the KEYLST used for the compiler questions:

```
((ST "ore and redefine " KEYLST ("" (F . "orget
exprs"))
 (S . "ame as last time")
 (F . "File only")
 (T . "o terminal")
 1
 2
 (Y . "es")
 (N . "o"))
```

When ASKUSER is called with this KEYLST, and the user types an S, two keys are matched: ST and S.

The user can then type a T, which matches only the ST key, or confirm the S key by typing a ^{cr} or space. If the user confirms the S key, ASKUSER prompts with "ame as last time", and returns S as its value. (Note that the confirming character is not included in the value.) If the user types a T, ASKUSER prompts with "ore and redefine", and makes ("" (F . "orget exprs")) be the new KEYLST, and waits for more input. The user can then type an F, or confirm the "" (which essentially starts out with all of its characters matched). If he confirms the "", ASKUSER returns ST as its value the result of packing ST and "". If he types F, ASKUSER prompts with "orget exprs", and waits for confirmation again. If the user then confirms, ASKUSER returns STF, the result of packing ST and F.

At any point the user can type a ? and be prompted with the possible responses. For example, if the user types S and then ?, ASKUSER will type:

```
STore and redefine Forget exprs
STore and redefine
Same as last time
```

Options

KEYLST	When a key is complete, if the value of the KEYLST option is not NIL, this value becomes the new KEYLST and the process recurses. Otherwise, the key terminates a path through the original, top-level KEYLST, and ASKUSER returns the indicated value.
CONFIRMFLG	If T, the key must be confirmed with either a carriage return or a space. If the value of CONFIRMFLG is a list, the confirming character may be any member of the list.
PROMPTCONFIRMFLG	If T, whenever confirmation is required, the user is prompted with the string [confirm].
NOCASEFLG	If T, says do not perform case independent matching on alphabetic characters. If NIL, do perform case independent matching, i.e. "A" matches with "a" and vice versa.

RETURN If non-NIL, EVAL of the value of the RETURN option is returned as the value of ASKUSER. Note that different RETURN options can be specified for different keys. The variable ANSWER is bound in ASKUSER to the list of keys that have been matched. In other words, RETURN (PACK ANSWER) would be equivalent to what ASKUSER normally does.

NOECHOFLG If non-NIL, characters that are matched (or automatically supplied as a result of typing $ (escape) or confirming) are not echoed, nor is the confirming character, if any. The value of NOECHOFLG is automatically NIL when ASKUSER is reading from a file or string. The decision about whether or not to echo a character that matches several keys is determined by the value of the NOECHOFLG option for the first key.

EXPLAINSTRING If the value of the EXPLAINSTRING option is non-NIL, its value is printed when the user types a ?, rather than KEY + PROMPTSTRING. EXPLAINSTRING enables more elaborate explanations in response to a ? than what the user sees when he is prompted as a result of simply completing keys. For example: One of the entries on the KEYLST used by ADDTOFILES? is:

```
(] "Nowherecr" NOECHOFLG T
 EXPLAINSTRING "] - nowhere, item is marked as a
dummycr")
```

When the user types], ASKUSER just prints Nowherecr, i.e., the] is not echoed. If the user types ?, the explanation corresponding to this entry will be:

```
 ] - nowhere, item is marked as a dummy
```

KEYSTRING If non-NIL, characters that are matched are echoed as though the value of KEYSTRING were used in place of the key. KEYSTRING is also used for computing the value returned. The main reason for this feature is to enable echoing in lowercase.

PROMPTON If non-NIL, PROMPTSTRING is printed only when the key is confirmed with a member of the value of PROMPTON.

COMPLETEON When a confirming character is typed, the N characters that are automatically supplied, as specified in case (4), are echoed only when the key is confirmed with a member of the value of PROMPTON.

The PROMPTON and COMPLETEON options enable the user to construct a KEYLST which will cause ASKUSER to emulate the action of the TENEX exec. The protocol followed by the TENEX exec is that the user can type as many characters as he likes in specifying a command. The command can be completed with a carriage return or space, in which case no further output is forthcoming, or with a $ (escape), in which case the rest of the characters in the command are echoed, followed by some prompting information. The following KEYLST would handle the TENEX COPY and CONNECT comands:

```
                    ((COPY " (FILE LIST) "
                           PROMPTON ($)
                           COMPLETEON ($)
                           CONFIRMFLG ($))

                      (CONNECT " (TO DIRECTORY) "
```

```
                              PROMPTON ($)
                              COMPLETEON ($)
                              CONFIRMFLG ($)))
```

AUTOCOMPLETEFLG   If the value of the AUTOCOMPLETEFLG option is not NIL, ASKUSER will automatically supply unambiguous characters whenever it can, i.e., ASKUSER acts as though $ (escape) were typed after each character (except that it does not ring the bell if there are no unambiguous characters).

MACROCHARS   value is a list of dotted pairs of form (CHARACTER . FORM). When CHARACTER is typed, and it does not match any of the current keys, FORM is evaluated and nothing else happens, i.e. the matching process stays where it is. For example, ? could have been implemented using this option. Essentially MACROCHARS provides a read macro facility while inside of ASKUSER (since ASKUSER does READC's, read macros defined via the readtable are never invoked).

EXPLAINDELIMITER   **value is what is printed to delimit explanation in** response to ?. Initially a carriage return, but can be reset, e.g. to a comma, for more linear output.

Operation

All input operations are executed with the terminal table in the variable ASKUSERTTBL, in which the following is true:

-

    (CONTROL T) has been executed (see the Line-Buffering section of Chapter 30), so that ASKUSER can interact with the user after each character is typed

    (ECHOMODE NIL) has been executed (see the Terminal Control Functions section of Chapter 30), so that ASKUSER can decide after it reads a character whether or not the character should be echoed, and with what, e.g. unacceptable inputs are never echoed.

As each character is typed, it is matched against KEYLST, and appropriate echoing and/or prompting is performed. If the user types an unacceptable character, ASKUSER simply rings the bell and allows him to try again.

At any point, the user can type ? and receive a list of acceptable responses at that point (generated from KEYLST), or type a Control-A, Control-Q, Control-X, or delete, which causes ASKUSER to reinitialize, and start over.

Note that ?, Control-A, Control-Q, and Control-X will not work if they are acceptable inputs, i.e., they match one of the keys on KEYLST. Delete will not work if it is an interrupt character, in which case it is not seen by ASKUSER.

When an acceptable sequence is completed, ASKUSER returns the indicated value.

Completing a Key

The decision about when a key is complete is more complicated than simply whether or not all of its characters have been matched. In the compiler questions example above, all of the characters in the S key are matched as soon as the S has been typed, but until the next character is typed, ASKUSER does not know whether the S completes the S key, or is simply the first character in the ST key. Therefore, a key is considered to be complete when:

1. All of its characters have been matched and it is the only key left, i.e., there are no other keys for which this key is a substring.

2. All of its characters have been matched and a confirming character is typed.

3. All of its characters have been matched, and the value of the CONFIRMFLG option is NIL, and the value of the KEYLST option is not NIL, and the next character matches one of the keys on the value of the KEYLST option.

4. There is only one key left and a confirming character is typed. Note that if the value of CONFIRMFLG is T, the key still has to be confirmed, regardless of whether or not it is complete. For example, if the first entry in the above example were instead

   ```
   (ST "ore and redefine " CONFIRMFLG T KEYLST ("" (F . "orget
   exprs"))
   ```

   and the user wanted to specify the STF path, he would have to type ST, then   confirm before typing F, even though the ST completed the ST key by the rule in Case 1. However, he would be prompted with ore and redefine as soon as he typed the T, and completed the ST key.

Case 2 says that confirmation can be used to complete a key in the case where it is a substring of another key, even where the value of CONFIRMFLG is NIL. In this case, the confirming character doubles as both an indicator that the key is complete, and also to confirm it, if necessary. This situation corresponds to typing Scr in the above example.

Case 3 says that if there were another entry whose key was STX in the above example, so that after the user typed ST, two keys, ST and STX, were still active, then typing F would complete the ST key, because F matches the (F . "orget exprs") entry on the value of the KEYLST option of the ST entry. In this case, ore and redefine would be printed before the F was echoed.

Finally, Case 4 says that the user can use confirmation to specify completion when only one key is left, even when all of its characters have not been matched. For example, if the first key in the above example were STORE, the user could type ST and then confirm, and ORE would be echoed, followed by whatever prompting was specified. In this case, the confirming character also confirms the key if necessary, so that no further action is required, even when the value of CONFIRMFLG is T.

Case 4 permits the user not to have to type every character in a key when the key is the only one left. Even when there are several active keys, the user can type <escape> to specify the next N>0 common characters among the currently active keys. The effect is exactly the same as though these characters

had been typed. If there are no common characters in the active keys at that point, i.e. `N = 0`, the `$` is treated as an incorrect input, and the bell is rung. For example, if `KEYLST` is `(CLISPFLG CLISPIFYPACKFLG CLISPIFTRANFLG)`, and the user types `C` followed by `$`, `ASKUSER` will supply the `L`, `I`, `S`, and `P`. The user can then type `F` followed by a carriage return or space to complete and confirm `CLISPFLG`, as per Case 4, or type `I`, followed by `$`, and `ASKUSER` will supply the `F`, etc. Note that the characters supplied do not have to correspond to a terminal segment of any of the keys. Note also that the `$` does not confirm the key, although it may complete it in the case that there is only one key active.

If the user types a confirming character when several keys are left, the next N>0 common characters are still supplied, the same as with `$`. However, `ASKUSER` assumes the intent was to complete a key, i.e., Case 4 is being invoked. Therefore, after supplying the next N characters, the bell is rung to indicate that the operation was not completed. In other words, typing a confirming character has the same effect as typing an `$` in that the next N common characters are supplied. Then, if there is only one key left, the key is complete (Case 4) and confirmation is not required. If the key is not the only key left, the bell is rung.

Special Keys

      `&`   This can be used as a key to match with any single character, provided the character does not match with some other key at that level. For the purposes of echoing and returning a value, the effect is the same as though the character that were matched actually appeared as the key.

    `<escape>`   This can be used as a key to match with the result of a single call to `READ`. For example, if the `KEYLST` were:

```
((COPY " (FILE LIST) "
        PROMPTON ($)
        COMPLETEON ($)
        CONFIRMFLG ($)
        KEYLST (($ NIL RETURN ANSWER))))
```

        then if the user typed `COP FOOcr`, `(COPY  FOO)` would be returned as the value of `ASKUSER`. One advantage of using `$`, rather than having the calling program perform the `READ`, is that the call to `READ` from inside `ASKUSER` is `ERRORSET` protected, so that the user can back out of this path and reinitialize `ASKUSER`, e.g. to change from a `COPY` command to a `CONNECT` command, simply by typing Control-E.

  Escape Escape   This can be used as a key to match with the result of a single call to `READLINE`.

      A list   A list can be used as a key, in which case the list/form is evaluated and its value "matches" the key. This feature is provided primarily as an escape hatch for including arbitrary input operations as part of an `ASKUSER` sequence. For example, the effect of `$$` (escape, escape) could be achieved simply by using `(READLINE T)` as a key.

    `""`   The empty string can be used as a key. Since it has no characters, all of its characters are automatically matched. `""` essentially functions as a place marker. For example, one of the entries on the `KEYLST` used by `ADDTOFILES?` is:

```
("" "File/list:  "
EXPLAINSTRING "a  file  name  or  name  of  a
function list"
```

`KEYLST ($))`

Thus, if the user types a character that does not match any of the other keys on the `KEYLST`, then the character completes the `""` key, by virtue of case (4), since the character will match with the `$` in the inner `KEYLST`. `ASKUSER` then prints File/list: before echoing the character, then calls `READ`. The character will be read as part of the `READ`. The value returned by `ASKUSER` will be the value of the `READ`.

Note: For Escape, Escape Escape, or a list, if the last character read by the input operation is a separator, the character is treated as a confirming character for the key. However, if the last character is a break character, it will be matched against the next key.

Startup Protocol and Typeahead

Interlisp permits and encourages the user to typeahead; in actual practice, the user frequently does this. This presents a problem for `ASKUSER`. When `ASKUSER` is entered and there has been typeahead, was the input intended for `ASKUSER`, or was the interaction unanticipated, and the user simply typing ahead to some other program, e.g. the programmer's assistant? Even where there was no typeahead, i.e., the user starts typing after the call to `ASKUSER`, the question remains of whether the user had time to see the message from `ASKUSER` and react to it, or simply began typing ahead at an inauspicious moment. Thus, what is needed is an interlock mechanism which warns the user to stop typing, gives him a chance to respond to the warning, and then allows him to begin typing to `ASKUSER`.

Therefore, when `ASKUSER` is first entered, and the interaction is to take place with a terminal, and typeahead to `ASKUSER` is not permitted, the following protocol is observed:

1. If there is typeahead, ASKUSER clears and saves the input buffers and rings the bell to warn the user to stop typing. The buffers will be restored when ASKUSER completes operation and returns.

2. If MESS, the message to be printed on entry, is not NIL (the typical case), ASKUSER then prints MESS if it is a string, otherwise CAR of MESS, if MESS is a list.

3. After printing MESS or CAR of MESS, ASKUSER waits until the output has actually been printed on the terminal to make sure that the user has actually had a chance to see the output. This also give the user a chance to react. ASKUSER then checks to see if anything additional has been typed in the intervening period since it first warned the user in (1). If something has been typed, ASKUSER clears it out and again rings the bell. This latter material, i.e., that typed between the entry to ASKUSER and this point, is discarded and will not be restored since it is not certain whether the user simply reacted quickly to the first warning (bell) and this input is intended for ASKUSER, or whether the user was in the process of typing ahead when the call to ASKUSER occurred, and did not stop typing at the first warning, and therefore this input is a continuation of input intended for another program.

   Anything typed after (3) is considered to be intended for ASKUSER, i.e., once the user sees MESS or CAR of MESS, he is free to respond. For example, UNDO (see Chapter 13) calls ASKUSER when the number of undosaves are exceeded for an event with MESS = (LIST NUMBER-UNDOSAVES "undosaves, continue saving"). Thus, the user can type a response as soon as NUMBER-UNDOSAVES is typed.

4. ASKUSER then types the rest of MESS, if any.

5. Then ASKUSER goes into a wait loop until something is typed. If WAIT, the wait time, is not NIL, and nothing is typed in WAIT seconds, ASKUSER will type "..." and treat the elements of DEFAULT, the default value, as a list of characters, and begin processing them exactly as though they had been typed. If the user does type anything within WAIT seconds, he can then wait as long as he likes, i.e., once something has been typed, ASKUSER will not use the default value specified in DEFAULT.

   If the user wants to consider his response for more than WAIT seconds, and does not want ASKUSER to default, he can type a carriage return or a space, which are ignored if they are not specified as acceptable inputs by KEYLST (see below) and they are the first thing typed.

   If the calling program knows that the user is expecting an interaction with ASKUSER, e.g., another interaction preceded this one, it can specify in the call to ASKUSER that typeahead is permitted. In this case, ASKUSER simply notes whether there is any typeahead, then prints MESS and goes into a wait loop as described above.

   If there is typeahead that contains unacceptable input, ASKUSER will assume that the typeahead was not intended for ASKUSER, and will restore the typeahead when it completes operation and returns.

6. Finally, if the interaction is not with the terminal, i.e., the optional input file/string is specified, ASKUSER simply prints MESS and begins reading from the file/string.

TTYIN Display Typein Editor

`TTYIN` is an Interlisp function for reading input from the terminal. It features altmode completion, spelling correction, help facility, and fancy editing, and can also serve as a glorified free text input function. This document is divided into two major sections: how to use `TTYIN` from the user's point of view, and from the programmer's.

`TTYIN` exists in implementations for Interlisp-10 and Interlisp-D. The two are substantially compatible, but the capabilities of the two systems differ (Interlisp-D has a more powerful display and allows greater access to the system primitives needed to control it effectively; it also has a mouse, greatly reducing the need for keyboard-oriented editing commands). Descriptions of both are included in this document for completeness, but Interlisp-D users may find large sections irrelevant.

Entering Input With TTYIN

There are two major ways of using `TTYIN`: set `LISPXREADFN` to `TTYIN`, so the `LISPX` executive uses it to obtain input; and call `TTYIN` from within a program to gather text input. Mostly the same rules apply to both; places where it makes a difference are mentioned below.

The following characters may be used to edit your input, independent of what kind of terminal you are on. The more `TTYIN` knows about your terminal, of course, the nicer some of these will behave. Some functions are performed by one of several characters; any character that you happen to have assigned as an interrupt character will, of couse, not be read by `TTYIN`. There is a (somewhat inelegant) way of changing which characters perform which functions, described under `TTYINREADMACROS` later on.

Control-A **BACKSPACE** **DELETE**	Deletes a character. At the start of the second or subsequent lines of your input, deletes the last character of the previous line.
Control-W	Deletes a "word". Generally this means back to the last space or parenthesis.
Control-Q	Deletes the current line, or if the current line is blank, deletes the previous line.
Control-R	Refreshes the current line. Two in a row refreshes the whole buffer (when doing multi-line input).
ESCAPE	Tries to complete the current word from the spelling list provided to `TTYIN`, if any. In the case of ambiguity, completes as far as is uniquely determined, or rings the bell. For `LISPX` input, the spelling list may be `USERWORDS` (see discussion of `TTYINCOMPLETEFLG`.
	Interlisp-10 only: If no spelling list was provided, but the word begins with a "<", tries directory name completion (or filename completion if there is already a matching ">" in the current word).

?	If typed in the middle of a word will supply alternative completions from the SPLST argument to TTYIN (if any). ?ACTIVATEFLG (see the Assorted Flags section below) must be true to enable this feature.
Control-Y	Escapes to a Lisp user exec, from which you may return by the command OK. However, when in READ mode and the buffer is non-empty, Control-Y is treated as Lisp's unquote macro instead, so you have to use meta-Control-Y (below) to invoke the user exec.
LF in Interlisp-10	Retrieves characters from the previous non-empty buffer when it is able to; e.g., when typed at the beginning of the line this command restores the previous line you typed at TTYIN; when typed in the middle of a line fills in the remaining text from the old line; when typed following ↑Q or ↑W restores what those commands erased.
;	If typed as the first character of the line means the line is a comment; it is ignored, and TTYIN loops back for more input. Note: The exact behaviour of this character is determined by the value of TTYINCOMMENTCHAR (see the Assorted Flags section below).
Control-X	Goes to the end of your input (or end of expression if there is an excess right parenthesis) and returns if parentheses are balanced, beeps if not.

During most kinds of input, TTYIN is in "autofill" mode: if a space is typed near the right margin, a carriage return is simulated to start a new line. In fact, on cursor-addressable displays, lines are always broken, if possible, so that no word straddles the end of the line. The "pseudo-carriage return" ending the line is still read as a space, however; i.e., the program keeps track of whether a line ends in a carriage return or is merely broken at some convenient point. You won't get carriage returns in your strings unless you explicitly type them.

Mouse Commands

The mouse buttons are interpreted as follows during TTYIN input:

LEFT	Moves the caret to where the cursor is pointing. As you hold down LEFT, the caret moves around with the cursor; after you let up, any typein will be inserted at the new position.
MIDDLE	Like LEFT, but moves only to word boundaries.
RIGHT	Deletes text from the caret to the cursor, either forward or backward. While you hold down RIGHT, the text to be deleted is complemented; when you let up, the text actually goes away. If you let up outside the scope of the text, nothing is killed (this is how to "cancel" the command). This is roughly the same as CTRL-RIGHT with no initial selection (below).

If you hold down CTRL and/or SHIFT while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. You make a selection by bugging LEFT (to select a character) or MIDDLE (to select a word), and optionally extend the selection either left or right using RIGHT. While you are doing this, the caret does not move, but your selected text is highlighted in a manner indicating what is about to happen. When you have made your selection (all mouse buttons up now), lift up on CTRL and/or SHIFT and the action you have selected will occur, which is:

SHIFT The selected text as typein at the caret. The text is highlighted with a broken underline during selection.

CTRL Delete the selected text. The text is complemented during selection.

CTRL–SHIFT Combines the above: delete the selected text and insert it at the caret. This is how you move text about.

You can cancel a selection in progress by pressing LEFT or MIDDLE as if to select, and moving outside the range of the text.

The most recent text deleted by mouse command can be inserted at the caret by typing Middle-blank key (on the Xerox 1132) or the Open key (on the Xerox 1108). This is the same key that retrieves the previous buffer when issued at the end of a line.

Display Editing Commands

On terminals with a meta key: In Interlisp-10, TTYIN reads from the terminal in binary mode, allowing many more editing commands via the meta key, in the style of TVEDIT commands. Note that due to Tenex's unfortunate way of handling typeahead, it is not possible to type ahead edit commands before TTYIN has started (i.e., before its prompt appears), because the meta bit will be thrown away. Also, since Escape has numerous other meanings in Lisp and even in TTYIN (for completion), this is not used as a substitute for the meta key.

In Interlisp-D: Users will probably have little use for most of these commands, as cursor positioning can often be done more conveniently, and certainly more obviously, with the mouse. Nevertheless, some commands, such as the case changing commands, can be useful. The <bottom-blank> key can be used as an meta key if you perform (METASHIFT T) (see Chapter 30). Alternatively, you can use the variable EDITPREFIXCHAR as described in the next paragraph.

On display terminals without a meta key: If you want to type any of these commands, you need to prefix them with the "edit prefix" character. Set the variable EDITPREFIXCHAR to the character code of the desired prefix char. Type the edit prefix twice to give an "meta-escape" command. Some users of the TENEX TVEDIT program like to make escape (33Q) be the edit prefix, but this makes it somewhat awkward to ever use escape completion. EDITPREFIXCHAR is initially NIL.

On hardcopy terminals without a meta key: You probably want to ignore this section, since you won't be able to see what's going on when you issure edit commands; there is no attempt made to echo anything reasonable.

In the descriptions below, "current word" means the word the cursor is under, or if under a space, the previous word. Currently parentheses are treated as spaces, which is usually what you want, but can occasionally cause confusion in the word deletion commands. Most commands can be preceded by numbers or escape (means infinity), only the first of which requires the meta key (or the edit prefix). Some commands also accept negative arguments, but some only look at the magnitude of the arg.

Most of these commands are taken from the display editors `TVEDIT` and/or `E`, and are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands:

■ **Meta-DELETE**
  Meta-BS
  Meta-<   Back up one (or n) characters.

  Meta-SPACE
  Meta->   Moves forward one (or n) characters.

  Meta-^   Moves up one (or n) lines.

  Meta-lf   Moves down one (or n) lines.

  Meta-(Moves back one (or n) words.

  Meta-)   Moves ahead one (or n) words.

  Meta-TAB   Moves to end of line; with an argument moves to nth end of line; **Meta-ESC-TAB** goes to end of buffer.

Control-Meta-L   Moves to start of line (or nth previous, or start of buffer).

  Meta-{
  Meta-}   Go to start and end of buffer, respectively.

  Meta-[Moves to beginning of the current list, where cursor is currently under an element of that list or its closing paren. (See also the auto-parenthesis-matching feature below under "Flags".)

  Meta-]   Moves to end of current list.

  Meta-Sx   Skips ahead to next (or nth) occurrence of character x, or rings the bell.

  Meta-Bx   Backward search.

Buffer Modification Commands:

  Meta-Zx   Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command yet.

  Meta-A
  Meta-R   Repeat the last S, B or Z command, regardless of any intervening input (note this differs from TEdit's A command).

  Meta-K   Kills the character under the cursor, or n chars starting at the cursor.

  Meta-CR   When the buffer is empty is the same as **LF**, i.e. restores buffer's previous contents. Otherwise is just like a **CR** (except that it also terminates an insert). Thus, **Meta-CR CR** will repeat the previous input (as will **LF CR** without the meta key).

Meta-O   Does "Open line", inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.

Meta-T   Transposes the characters before and after the cursor. When typed at the end of a line, transposes the previous two characters. Refuses to handle funny cases, such as tabs.

Meta-G   Grabs the contents of the previous line from the cursor position onward. Meta-nG grabs the nth previous line.

Meta-L   Lowercases current word, or n words on line. **Meta-ESC-L** lowercases the rest of the line, or if given at the end of line lowercases the entire line.

Meta-U   Uppercases analogously.

Meta-C   Capitalize. If you give it an argument, only the first word is capitalized; the rest are just lowercased.

Control-Meta-Q   Deletes the current line. **Control-Meta-ESC-Q** deletes from the current cursor position to the end of the buffer. No other arguments are handled.

Control-Meta-W   Deletes the current word, or the previous word if sitting on a space.

Meta-J   "Justify" this line. This will break it if it is too long, or move words up from the next line if too short. Will not join to an empty line, or one starting with a tab (both of which are interpreted as paragraph breaks). Any new line breaks it introduces are considered spaces, not carriage returns. **Meta-nJ** justifies n lines.

The linelength is defined as TTYJUSTLENGTH, ignoring any prompt characters at the margin. If TTYJUSTLENGTH is negative, it is interpreted as relative to the right margin. TTYJUSTLENGTH is initially -8 in Interlisp-D, 72 in Interlisp-10.

Meta-ESC-F   "Finishes" the input, regardless of where the cursor is. Specifically, it goes to the end of the input and enters a **CR, control-Z** or "", depending on whether normal, REPEAT or READ input is happening. Note that a "" won't necessarily end a READ, but it seems likely to in most cases where you would be inclined to use this command, and makes for more predictable behavior.

Miscellaneous Commands:

Meta-P   Interlisp-D: Prettyprint buffer. Clears the buffer and reprints it using prettyprint. If there are not enough right parentheses, it will supply more; if there are too many, any excess remains unprettyprinted at the end of the buffer. May refuse to do anything if there is an unclosed string or other error trying to read the buffer.

Meta-N   Refresh line. Same as **Control-R**. Meta-ESC-N refreshes the whole buffer; Meta-nN refreshes n lines. Cursor movement in TTYIN depends on TTYIN being the only source of output to the screen; if you do a **Control-T**, or a system message appears, or line noise occurs, you may need to refresh the line for best results. In Interlisp-10, if for some reason your terminal falls out of binary mode (e.g. can happen when returning to a Lisp running in a lower fork), **Meta-<anything>** is unreadable, so you'd have to type **Control-R** instead.

Control-Meta-Y   Gets user exec. Thus, this is like regular **Control-Y**, except when doing a READ (when control-Y is a read macro and hence does not invoke this function).

Control-Meta-ESC-Y Gets a user exec, but first unreads the contents of the buffer from the cursor onward. Thus if you typed at TTYIN something destined for the Lisp executive, you can do **Control-Meta-L-ESC-Control-Y** and give it to Lisp.

Meta-← Adds the current word to the spelling list USERWORDS. With zero arg, removes word. See TTYINCOMPLETEFLG (see the Assorted Flags section below).

Note to Datamedia, Heath users: In addition to simple cursor movement commands and insert/delete, TTYIN uses the display's cursor-addressing capability to optimize cursor movements longer than a few characters, e.g. **Meta-TAB** to go to the end of the line. In order to be able to address the cursor, TTYIN has to know where it is to begin with. Lisp keeps track of the current print position within the line, but does not keep track of the line on the screen (in fact, it knows precious little about displays, much like Tenex). Thus, TTYIN establishes where it is by forcing the cursor to appear on the last line of the screen. Ordinarily this is the case anyway (except possibly on startup), but if the cursor happens to be only halfway down the screen at the time, there is a possibly unsettling leap of the cursor when TTYIN starts.

Using TTYIN for Lisp Input

When TTYIN is loaded, or a sysout containing TTYIN is started up, the function SETREADFN is called. If the terminal is a display, it sets LISPXREADFN (see Chapter 13) to be TTYINREAD. If the terminal is not a display terminal, SETREADFN will set the variable to READ. (SETREADFN 'READ) will also set it to READ.

There are two principal differences between TTYINREAD and READ: (1) parenthesis balancing. The input does not activate on an exactly balancing right paren/bracket unless the input started with a paren/bracket, e.g., USE (FOO) FOR (FIE) will all be on one line, terminated by **CR**; and (2) read macros.

In Interlisp-10, TTYIN does not use a read table (TTYIN behaves as though using the default initial Lisp terminal input readtable), so read macros and redefinition of syntax characters are not supported; however, " ' " (QUOTE) and "Control-Y" (EVAL) are built in, and a simple implementation of ? and ?= is supplied. Also, the TTYINREADMACROS facility described below can supply some of the functionality of immediate read macros in the editor.

In Interlisp-D, read macros are (mostly) supported. Immediate read macros take effect only if typed at the end of the input (it's not clear what their semantics should be elsewhere).

Useful Macros

There are two useful edit macros that allow you to use TTYIN as a character editor: (1) ED loads the current expression into the ttyin buffer to be edited (this is good for editing comments and strings). Input is terminated in the usual way (by typing a balancing right programmer's assistant command FIX will load the buffer with the event's input, rather than calling the editor. If you really wanted the

Interlisp editor for your fix, you can say `FIX EVENT – TTY:` once you got `TTYIN`'s version to force you into the editor.

Programming With TTYIN

(**TTYIN** PROMPT SPLST HELP OPTIONS ECHOTOFILE TABS UNREADBUF RDTBL)
[Function]

> `TTYIN` prints `PROMPT`, then waits for input. The value returned in the normal case is a list of all atoms on the line, with comma and parens returned as individual atoms; `OPTIONS` may be used to get a different kind of value back.

> `PROMPT` is an atom or string (anything else is converted to a string). If `NIL`, the value of `DEFAULTPROMPT`, initially "** ", will be used. If `PROMPT` is `T`, no prompt will be given. `PROMPT` may also be a dotted pair (PROMPT1 . PROMPT2), giving the prompt for the first and subsequent (or overflow) lines, each prompt being a string/atom or `NIL` to denote absence of prompt. The default prompt for overflow lines is "...". Note that rebinding `DEFAULTPROMPT` gives a convenient way to affect all the "ordinary" prompts in some program module.

> `SPLST` is a spelling list, i.e., a list of atoms or dotted pairs (SYNONYM . ROOT). If supplied, it is used to check and correct user responses, and to provide completion if the user types escape. If `SPLST` is one of the Lisp system spelling lists (e.g., `USERWORDS` or `SPELLINGS3`), words that are escape-completed get moved to the front, just as if a `FIXSPELL` had found them. Autocompletion is also performed when user types a break character (cr, space, paren, etc), unless one of the "nofixspell" options below is selected; i.e., if the word just typed would uniquely complete by escape, `TTYIN` behaves as though escape had been typed.

> `HELP`, if non-`NIL`, determines what happens when the user types `?` or `HELP`. If `HELP = T`, program prints back `SPLST` in suitable form. If `HELP` is any other litatom, or a string containing no spaces, it performs (DISPLAYHELP HELP). Anything else is printed as is. If `HELP` is `NIL`, `?` and `HELP` are treated as any other atoms the user types. [`DISPLAYHELP` is a user-supplied function, initially a noop; systems with a suitable `HASH` package, for example, have defined it to display a piece of text from a hashfile associated with the key `HELP`.]

> `OPTIONS` is an atom or list of atoms chosen from among the following:

> NOFIXSPELL   Uses `SPLST` for `HELP` and Escape completion, but does not attempt any `FIXSPELL`ing. Mainly useful if `SPLST` is incomplete and the caller wants to handle corrections in a more flexible way than a straight `FIXSPELL`.

> MUSTAPPROVE   Does spelling correction, but requires confirmation.

> CRCOMPLETE   Requires confirmation on spelling correction, but also does autocompletion on <cr> (i.e. if what user has typed so far uniquely identifies a member of `SPLST`, completes it). This allows you to have the benefits of autocompletion and still allow new words to be typed.

DIRECTORY (only if `SPLST` = `NIL`) Interprets Escape to mean directory name completion [Interlisp-10 only].

USER Like `DIRECTORY`, but does username completion. This is identical to `DIRECTORY` under Tenex [Interlisp-10 only].

FILE (only if `SPLST` = `NIL`) Interprets Escape to mean filename completion [Sumex and Tops20 only].

FIX If response is not on, or does not correct to, `SPLST`, interacts with user until an acceptable response is entered. A blank line (returning `NIL`) is always accepted. Note that if you are willing to accept responses that are not on `SPLST`, you probably should specify one of the options `NOXFISPELL`, `MUSTAPPROVE` or `CRCOMPLETE`, lest the user's new response get `FIXSPELL`ed away without their approval.

STRING Line is read as a string, rather than list of atoms. Good for free text.

NORAISE Does not convert lower case letters to upper case.

NOVALUE For use principally with the `ECHOTOFILE` arg (below). Does not compute a value, but returns `T` if user typed anything, `NIL` if just a blank line.

REPEAT For multi-line input. Repeatedly prompts until user types Control-Z (as in Tenex sndmsg). Returns one long list; with `STRING` option returns a single string of everything typed, with carriage returns (`EOL`) included in the string.

TEXT Implies `REPEAT`, `NORAISE`, and `NOVALUE`. Additionally, input may be terminated with Control-V, in which case the global flag `CTRLVFLG` will be set true (it is set to `NIL` on any other termination). This flag may be utilized in any way the caller desires.

COMMAND Only the first word on the line is treated as belonging to `SPLST`, the remainder of the line being arbitrary text; i.e., "command format". If other options are supplied, `COMMAND` still applies to the first word typed. Basically, it always returns (`CMD` . `REST-OF-INPUT`), where `REST-OF-INPUT` is whatever the other options dictate for the remainder. E.g. `COMMAND NOVALUE` returns (`CMD`) or (`CMD` . `T`), depending on whether there was further input; `COMMAND STRING` returns (`CMD` . `"REST-OF-INPUT"`). When used with `REPEAT`, `COMMAND` is only in effect for the first line typed; furthermore, if the first line consists solely of a command, the `REPEAT` is ignored, i.e., the entire input is taken to be just the command.

READ Parens, brackets, and quotes are treated a la `READ`, rather than being returned as individual atoms. Control characters may be input via the Control-Vx notation. Input is terminated roughly along the lines of `READ` conventions: a balancing or over-balancing right paren/bracket will activate the input, or <cr> when no parenthesis remains unbalanced. `READ` overrides all other options (except `NORAISE`).

LISPXREAD   Like READ, but implies that TTYIN should behave even more like READ, i.e., do NORAISE, not be errorset-protected, etc.

NOPROMPT   Interlisp-D only: The prompt argument is treated as usual, except that TTYIN assumes that the prompt for the first line has already been printed by the caller; the prompt for the first line is thus used only when redisplaying the line.

ECHOTOFILE if specified, user's input is copied to this file, i.e., TTYIN can be used as a simple text-to-file routine if NOVALUE is used. If ECHOTOFILE is a list, copies to all files in the list. PROMPT is not included on the file.

TABS is a special addition for tabular input. It is a list of tabstops (numbers). When user types a tab, TTYIN automatically spaces over to the next tabstop (thus the first tabstop is actually the second "column" of input). Also treats specially the characters * and "; they echo normally, and then automatically tab over.

UNREADBUF allows the caller to "preload" the TTYIN buffer with a line of input. UNREADBUF is a list, the elements of which are unread into the buffer (i.e., "the outer parentheses are stripped off") to be edited further as desired; a simple carriage return (or Control-Z for REPEAT input) will thus cause the buffer's contents to be returned unchanged. If doing READ input, the "PRIN2 names" of the input list are used, i.e., quotes and %'s will appear as needed; otherwise the buffer will look as though UNREADBUF had been PRIN1'ed. UNREADBUF is treated somewhat like READBUF, so that if it contains a pseudo-carriage return (the value of HISTSTR0), the input line terminates there.

Input can also be unread from a file, using the HISTSTR1 format: UNREADBUF = (<value of HISTSTR1> (FILE START . END)), where START and END are file byte pointers. This makes TTYIN a miniature text file editor.

RDTBL [Interlisp-D only] is the read table to use for READing the input when one of the READ options is given. A lot of character interpretations are hardwired into TTYIN, so currently the only effect this has is in the actual READ, and in deciding whether a character typed at the end of the input is an immediate read macro, for purposes of termination.

If the global variable TYPEAHEADFLG is T, or option LISPXREAD is given, TTYIN permits type-ahead; otherwise it clears the buffer before prompting the user.

Using TTYIN as a General Editor

The following may be useful as a way of outsiders to call TTYIN as an editor. These functions are currently only in Interlisp-D.

(**TTYINEDIT** *EXPRS WINDOW PRINTFN PROMPT*) [Function]

> This is the body of the edit macro EE. Switches the tty to *WINDOW*, clears it, prettyprints *EXPRS*, a list of expressions, into it, and leaves you in TTYIN to edit it as Lisp input. Returns a new list of expressions.

> If *PRINTFN* is non-NIL, it is a function of two arguments, EXPRS and FILE, which is called instead of PRETTYPRINT to print the expressions to the window (actually to a scratch file). Note that EXPRS is a list, so normally the outer parentheses should not be printed. *PRINTFN* = T is shorthand for "unpretty"; use PRIN2 instead of PRETTYPRINT.

> *PROMPT* determines what prompt is printed, if any. If T, no prompt is printed. If NIL, it defaults to the value of TTYINEDITPROMPT.

TTYINAUTOCLOSEFLG [Variable]

> If TTYINAUTOCLOSEFLG is true, TTYINEDIT closes the window on exit.

TTYINEDITWINDOW [Variable]

> If the WINDOW arg to TTYINEDIT is NIL, it uses the value of TTYINEDITWINDOW, creating it if it does not yet exist.

TTYINPRINTFN [Variable]

> The default value for PRINTFN in EE's call to TTYINEDIT.

(**SET.TTYINEDIT.WINDOW** *WINDOW*) [Function]

> Called under a RESETLST. Switches the tty to *WINDOW* (defaulted as in TTYINEDIT) and clears it. The window's position is left so that TTYIN will be happy with it if you now call TTYIN yourself. Specifically, this means positioning an integral number of lines from the bottom of the window, the way the top-level tty window normally is.

(**TTYIN.SCRATCHFILE**) [Function]

> Returns, possibly creating, the scratchfile that TTYIN uses for prettyprinting its input. The file pointer is set to zero. Since TTYIN does use this file, beware of multiple simultaneous use of the file.

?= Handler

In Interlisp, the ?= read macro displays the arguments to the function currently "in progress" in the typein. Since TTYIN wants you to be able to continue editing the buffer after a ?=, it processes this macro specially on its own, printing the arguments below your typein and then putting the cursor back where it was when ?= was typed. For users who want special treatment of ?=, the following hook exists:

TTYIN?=FN [Variable]

The value of this variable, if non-`NIL`, is a user function of one argument that is called when `?=` is typed. The argument is the function that `?=` thinks it is inside of. The user function should return one of the following:

`NIL`   Normal ?= processing is performed.

`T`   Nothing is done. Presumably the user function has done something privately, perhaps diddled some other window, or called `TTYIN.PRINTARGS` (below).

a list (`ARGS . STUFF`)   Treats `STUFF` as the argument list of the function in question, and performs the normal `?=` processing using it.

anything else   The value is printed in lieu of what `?=` normally prints.

At the time that `?=` is typed, nothing has been "read" yet, so you don't have the normal context you might expect inside a conventional readmacro. If the user function wants to examine the typed-in arguments being passed to the fn, however, it can call the function `TTYIN.READ?=ARGS`:

(**TTYIN.READ?=ARGS**) [Function]

When called inside `TTYIN?=FN` user function, returns everything between the function and the typing of ?= as a list (like an arglist). Returns `NIL` if ?= was typed immediately after the function name.

(**TTYIN.PRINTARGS** *FN ARGS ACTUALS ARGTYPE*) [Function]

Does the function/argument printing for `?=`. *ARGS* is an argument list, *ACTUALS* is a list of actual parameters (from the typein) to match up with args. *ARGTYPE* is a value of the function *ARGTYPE*; it defaults to (*ARGTYPE FN*).

Read Macros

When doing `READ` input in Interlisp-10, no Lisp-style read macros are available (but the ' and control-Y macros are built in). Principally because of the usefulness of the editor read macros (set by `SETTERMCHARS`), and the desire for a way of changing the meanings of the display editing commands, the following exists as a hack:

TTYINREADMACROS [Variable]

Value is a set of shorthand inputs useable during `READ` input. It is an alist of entries (`CHARCODE . SYNONYM`). If the user types the indicated character (the meta bit is denoted by the 200Q bit in the char code), `TTYIN` behaves as though the synonym character had been typed.

Special cases: 0 - the character is ignored; 200Q - pure meta bit; means to read another char and turn on its meta bit; 400Q - macro quote: read another char and use its original meaning. For example, if you have macros ((33Q . 200Q) (30Q . 33Q)), then Escape (33Q) will behave as an edit prefix, and control-X (30Q) will behave like Escape.

Note: currently, synonyms for meta commands are not well-supported, working only when the command is typed with no argument.

Slightly more powerful macros also can be supplied; they are recognized when a character is typed on an empty line, i.e., as the first thing after the prompt. In this case, the TTYINREADMACROS entry is of the form (CHARCODE T . RESPONSE) or (CHARCODE CONDITION . RESPONSE), where CONDITION is a list that evaluates true. If RESPONSE is a list, it is EVALed; otherwise it is left unevaluated. The result of this evaluation (or RESPONSE itself) is treated as follows:

NIL   The macro is ignored and the character reads normally, i.e., as though TTYINREADMACROS had never existed.

An integer   A character code, treated as above. Special case: -1 is treated like 0, but says that the display may have been altered in the evaluation of the macro, so TTYIN should reset itself appropriately.

Anything else   This TTYIN input is terminated (with a crlf) and returns the value of "response" (turned into a list if necessary). This is the principal use of this facility. The macro character thus stands for the (possibly computed) reponse, terminated if necessary with a crlf. The original character is not echoed.

Interrupt characters, of course, cannot be read macros, as TTYIN never sees them, but any other characters, even non-control chars, are allowed. The ability to return NIL allows you to have conditional macros that only apply in specified situations (e.g., the macro might check the prompt (LISPXID) or other contextual variables). To use this specifically to do immediate editor read macros, do the following for each edit command and character you want to invoke it with:

```
(ADDTOVAR TTYINREADMACROS (CHARCODE 'CHARMACRO?
EDITCOM)))
```

For example, (ADDTOVAR TTYINREADMACROS (12Q CHARMACRO? !NX)) will make linefeed do the !NX command. Note that this will only activate linefeed at the beginning of a line, not anywhere in the line. There will probably be a user function to do this in the next release.

Note that putting (12Q T . !NX) on TTYINREADMACROS would also have the effect of returning !NX from the READ call so that the editor would do an !NX. However, TTYIN would also return !NX outside the editor (probably resulting in a u.b.a. error, or convincing DWIM to enter the editor), and also the clearing of the output buffer (performed by CHARMACRO?) would not happen.

Assorted Flags

These flags control aspects of TTYIN's behavior. Some have already been mentioned. In Interlisp-D, the flags are all initially set to T.

TYPEAHEADFLG [Variable]

If true, `TTYIN` always permits typeahead; otherwise it clears the buffer for any but `LISPXREAD` input.

?**ACTIVATEFLG** [Variable]

If true, enables the feature whereby `?` lists alternative completions from the current spelling list.

SHOWPARENFLG [Variable]

If true, then whenever you are typing Lisp input and type a right parenthesis/bracket, `TTYIN` will briefly move the cursor to the matching parenthesis/bracket, assuming it is still on the screen. The cursor stays there for about 1 second, or until you type another character (i.e., if you type fast you'll never notice it). This feature was inspired by a similar `EMACS` feature, and turned out to be pretty easy to implement.

TTYINBSFLG [Variable]

Causes `TTYIN` to always physically backspace, even if you're running on a non-display (not a DM or Heath), rather than print \deletedtext\ (this assumes your hardcopy terminal or glass tty is capable of backspacing). If `TTYINBSFLG` is `LF`, then in addition to backspacing, `TTYIN` x's out the deleted characters as it backs up, and when you stop deleting, it outputs a linefeed to drop to a new, clean line before resuming. To save paper, this linefeed operation is not done when only a single character is deleted, on the grounds that you can probably figure out what you typed anyway.

TTYINRESPONSES [Variable]

An association list of special responses that will be handled by routines designated by the programmer. See "Special Responses", below.

TTYINERRORSETFLG [Variable]

[Interlisp-D only] If true, non-`LISPXREAD` inputs are errorset-protected (Control-E traps back to the prompt), otherwise errors propagate upwards. Initially `NIL`.

TTYINCOMMENTCHAR [Variable]

This variable affects the treatment of lines beginning with the comment character (usually ";"). If `TTYINCOMMENTCHAR` is a character code, and the first character on a line of typein is equal to `TTYINCOMMENTCHAR`, then the line is erased from the screen and no input function will see it. If `TTYINCOMMENTCHAR` is `NIL`, this feature is disabled. `TTYINCOMMENTCHAR` is initially `NIL`.

TTYINCOMPLETEFLG [Variable]

If true, enables Escape completion from `USERWORDS` during `READ` inputs. Details below.

USERWORDS (see Chapter 20) contains words you mentioned recently: functions you have defined or edited, variables you have set or evaluated at the executive level, etc. This happens to be a very convenient list for context-free escape completion; if you have recently edited a function, chances are good you may want to edit it again (typing "EF xx$") or type a call to it. If there is no completion for the current word from USERWORDS, the escape echoes as "$", i.e. nothing special happens; if there is more than one possible completion, you get beeped. If typed when not inside a word, Escape completes to the value of LASTWORD, i.e., the last thing you typed that the p.a. "noticed" (setting TTYINCOMPLETEFLG to 0 disables this latter feature), except that Escape at the beginning of the line is left alone (it is a p.a. command).

If you really wanted to enter an escape, you can, of course, just quote it with a control-V, like you can other control chars.

You may explicitly add words to USERWORDS yourself that wouldn't get there otherwise. To make this convenient online the edit command [←] means "add the current atom to USERWORDS" (you might think of the command as "pointing out this atom"). For example, you might be entering a function definition and want to "point to" one or more of its arguments or prog variables. Giving an argument of zero to this command will instead remove the indicated atom from USERWORDS.

Note that this feature loses some of its value if the spelling list is too long, for then the completion takes too long computationally and, more important, there are too many alternative completions for you to get by with typing a few characters followed by escape. Lisp's maintenance of the spelling list USERWORDS keeps the "temporary" section (which is where everything goes initially unless you say otherwise) limited to #USERWORDS atoms, initially 100. Words fall off the end if they haven't been used (they are "used" if FIXSPELL corrects to one, or you use <escape> to complete one).

Special Responses

There is a facility for handling "special responses" during any non-READ TTYIN input. This action is independent of the particular call to TTYIN, and exists to allow you to effectively "advise" TTYIN to intercept certain commands. After the command is processed, control returns to the original TTYIN call. The facility is implemented via the list TTYINRESPONSES.

TTYINRESPONSES [Variable]

TTYINRESPONSES is a list of elements, each of the form:

(COMMANDS RESPONSE-FORM OPTION)

COMMANDS is a single atom or list of commands to be recognized; RESPONSE-FORM is EVALed (if a list), or APPLYed (if an atom) to the command and the rest of the line. Within this form one can reference the free variables COMMAND (the command the user typed) and LINE (the rest of the line). If OPTION is the atom LINE, this means to pass the rest of line as a list; if it is STRING, this means to pass it as a string; otherwise, the command is only valid if there is nothing else on the line. If RESPONSE-FORM returns the atom IGNORE, it is not treated as a special response (i.e. the input is returned normally as the result of TTYIN).

Suggested use: global commands or options can be added to the toplevel value of TTYINRESPONSES. For more specialized commands, rebind TTYINRESPONSES to (APPEND NEWENTRIES TTYINRESPONSES) inside any module where you want to do this sort of special processing.

Special responses are not checked for during READ-style input.

Display Types

[This is not relevant in Interlisp-D]

TTYIN determines the type of display by calling DISPLAYTERMP, which is initially defined to test the value of the GTTYP jsys. It returns either NIL (for printing terminals) or a small number giving TTYIN's internal code for the terminal type. The types TTYIN currently knows about:

> 0 = glass tty (capable of deleting chars by backspacing, but little else)
>
> 1 = Datamedia
>
> 2 = Heath

Only the Datamedia has full editing power. DISPLAYTERMP has built into it the correct terminal types for Sumex and Stanford campus 20's: Datamedia = 11 on tenex, 5 on tops20; Heath = 18 on Tenex, 25 on tops20. You can override those values by setting the variable DISPLAYTYPES to be an association list associating the GTTYP value with one of these internal codes. For example, Sumex displays correspond to DISPLAYTYPES = ((11 . 1) (18 . 2)) [although this is actually compiled into DISPLAYTERMP for speed]. Any display terminal other than Datamedia and Heath can probably safely be assigned to "0" for glass tty.

To add new terminal types, you have to choose a number for it, add new code to TTYIN for it and recompile. The TTYIN code specifies what the capabilities of the terminal are, and how to do the primitive operations: up, down, left, right, address cursor, erase screen, erase to end of line, insert character, etc.

For terminals lacking a meta key (currently only Datamedias have it), set the variable EDITPREFIXCHAR to the ascii code of an edit "prefix" (i.e., anything typed preceded by the prefix is considered to have the meta bit on). If your EDITPREFIXCHAR is 33Q (Escape), you can type a real Escape by typing 3 of them (2 won't do, since that means "Meta-Escape", a legitimate argument to another command). You could also define an Escape synonym with TTYINREADMACROS if you wanted (but currently it doesn't work in filename completion). Setting EDITPREFIXCHAR for a terminal that is not equipped to handle the full range of editing functions (only the Heath and Datamedia are currently so equipped) is not guaranteed to work, i.e. the display will not always be up to date; but if you can keep track of what you're doing, together with an occasional control-R to help out, go right ahead.

Prettyprint

The standard way of printing out function definitions (on the terminal or into files) is to use
PRETTYPRINT.

(**PRETTYPRINT** *FNS PRETTYDEFLG* —)   [Function]

> *FNS* is a list of functions. If *FNS* is atomic, its value is used). The definitions of the
> functions are printed in a pretty format on the primary output file using the primary
> readtable. For example, if FACTORIAL were defined by typing

```
          (DEFINEQ (FACTORIAL [LAMBDA (N) (COND ((ZEROP N) 1)
          (T (ITIMES N (FACTORIAL (SUB1 N]

          (PRETTYPRINT '(FACTORIAL))would print out
          (FACTORIAL
            [LAMBDA (N)
              (COND
                ((ZEROP N)
                 1)
                (T (ITIMES N (FACTORIAL (SUB1 N])
```

> *PRETTYDEFLG* is T when called from PRETTYDEF (and hence MAKEFILE). Among other
> actions taken when this argument is true, PRETTYPRINT indicates its progress in writing
> the current output file: whenever it starts a new function, it prints on the terminal the
> name of that function if more than 30 seconds (real time) have elapsed since the last time
> it printed the name of a function.

> PRETTYPRINT operates correctly on functions that are BROKEN, BROKEN-IN, ADVISED, or
> have been compiled with their definitions saved on their property lists: it prints the
> original, pristine definition, but does not change the current state of the function. If a
> function is not defined but is known to be on one of the files noticed by the file package,
> PRETTYPRINT loads in the definition (using LOADFNS) and prints it (except when called
> from PRETTYDEF). If PRETTYPRINT is given an atom which is not the name of a
> function, but has a value, it prettyprints the value. Otherwise, PRETTYPRINT attempts
> spelling correction. If all fails, PRETTYPRINT returns (FN NOT PRINTABLE). Note that
> PRETTYPRINT will return (FN NOT PRINTABLE) if FN does not have an accessible expr
> definition, or if it doesn't have any definition at all.

(**PP** *FN$_1$ ... FN$_N$*)   [NLambda NoSpread Function]

> For prettyprinting functions to the terminal. PP calls PRETTYPRINT with the primary
> output file set to T and the primary read table set to T. The primary output file and
> primary readtable are restored after printing.

> (PP FOO) is equivalent to (PRETTYPRINT '(FOO)); (PP FOO FIE) is equivalent to
> (PRETTYPRINT '(FOO FIE)).

INTERLISP-D REFERENCE MANUAL

As described above, when PRETTYPRINT, and hence PP, is called with the name of a function that is not defined, but whose definition is on a file known to the file package, the definition is automatically read in and then prettyprinted. However, if the user does not intend on editing or running the definition, but simply wants to see the definition, the function PF described below can be used to simply copy the corresponding characters from the file to the terminal. This results in a savings in both space and time, since it is not necessary to allocate storage to actually read in the definition, and it is not necessary to re-prettyprint it (since the function is already in prettyprint format on the file).

> (**PF** *FN FROMFILES TOFILE*)   [NLambda NoSpread Function]
>
>> Copies the definition of *FN* found on each of the files in *FROMFILES* to *TOFILE*. If *TOFILE* = NIL, defaults to T. If *FROMFILES* = NIL, defaults to (WHEREIS FN NIL T) (see Chapter 17). The typical usage of PF is simply to type "PF FN".
>>
>> PF prints a message if it can't find a file on *FROMFILES*, or it can't find the function FN on a file.

When printing to the terminal, PF performs several transformations on the characters in the file that comprise the definition for FN:

1. Font information is stripped out (except in Interlisp-D, whose display supports multiple fonts)

2. Occurrences of the CHANGECHAR (see the Special Prettyprint Controls section below) are not printed

3. Since functions typically tend to be printed to a file with a larger linelength than when printing to a terminal, the number of leading spaces on each line is cut in half (unless PFDEFAULT is T; initially NIL)

4. Comments are elided, if **COMMENT**FLG is non-NIL (see the Comment Feature section below).

> (**SEE** *FROMFILE TOFILE*)   [NLambda NoSpread Function]
>
>> Copies all of the text from *FROMFILE* to *TOFILE* (defaults to T), processing all text as PF does. Used to display the contents of files on the terminal.

> (**PP*** *X*)   [NLambda NoSpread Function]
> (**PF*** *FN FROMFILES TOFILE*)   [NLambda NoSpread Function]
> (**SEE*** *FROMFILE TOFILE*)   [NLambda NoSpread Function]
>
>> These functions operate exactly like PP, PF, and SEE, except that they bind **COMMENT**FLG to NIL, so comments are printed in full.

While the function PRETTYPRINT prints entire function definitions, the function PRINTDEF can be used to print parts of functions, or arbitrary Interlisp structures:

(**PRINTDEF** *EXPR LEFT DEF TAILFLG FNSLST FILE*)   [Function]

>Prints the expression *EXPR* in a pretty format on *FILE* using the primary readtable. *LEFT* is the left hand margin (`LINELENGTH` determines the right hand margin). `PRINTDEF` initially performs (`TAB LEFT T`), which means to space to position *LEFT*, unless already beyond this position, in which case it does nothing.

>*DEF* = T means *EXPR* is a function definition, or a piece of one. If *DEF* = NIL, no special action is taken for LAMBDA's, PROG's, COND's, comments, CLISP, etc. DEF is NIL when PRETTYDEF calls PRETTYPRINT to print variables and property lists, and when PRINTDEF is called from the editor via the command PPV.

>*TAILFLG* = T means *EXPR* is interpreted as a tail of a list, to be printed without parentheses.

>*FNSLST* is for use for printing with multiple fonts (see Chapter 27). `PRINTDEF` prints occurrences of any function in the list `FNSLST` in a different font, for emphasis. `MAKEFILE` passes as `FNSLST` the list of all functions on the file being made.

Comment Feature

A facility for annotating Interlisp functions is provided in `PRETTYPRINT`. Any expression beginning with the atom * is interpreted as a comment and printed in the right margin. Example:

```
(FACTORIAL
   [LAMBDA (N)                  (* COMPUTES N!)
        (COND
          ((ZEROP N)            (* 0! = 1)
              1)
          (T                (* RECURSIVE DEFINITION:
                              N! = N*N-1!)
            (ITIMES N (FACTORIAL (SUB1 N])
```

These comments actually form a part of the function definition. Accordingly, * is defined as an nlambda nospread function that returns its argument, similar to `QUOTE`. When running an interpreted function, * is entered the same as any other Interlisp function. Therefore, comments should only be placed where they will not harm the computation, i.e., where a quoted expression could be placed. For example, writing

```
(ITIMES N (FACTORIAL (SUB1 N)) (* RECURSIVE
DEFINITION))
```

in the above function would cause an error when `ITIMES` attempted to multiply N, N-1!, and RECURSIVE.

For compilation purposes, * is defined as a macro which compiles into no instructions (unless the comment has been placed where it has been used for value, in which case the compiler prints an appropriate error message and compiles * as `QUOTE`). Thus, the compiled form of a function with

comments does not use the extra atom and list structure storage required by the comments in the source (interpreted) code. This is the way the comment feature is intended to be used.

A comment of the form (* E X) causes X to be evaluated at prettyprint time, as well as printed as a comment in the usual way. For example, (* E (RADIX 8)) as a comment in a function containing octal numbers can be used to change the radix to produce more readable printout.

The comment character * is stored in the variable COMMENTFLG. The user can set it to some other value, e.g. ";", and use this to indicate comments.

> **COMMENTFLG**   [Variable]
>
> > If CAR of an expression is EQ to COMMENTFLG, the expression is treated as a comment by PRETTYPRINT. COMMENTFLG is initialized to *. Note that whatever atom is chosen for COMMENTFLG should also have an appropriate function definition and compiler macro, for example, by copying those of *.

Comments are designed mainly for documenting listings. Therefore, when prettyprinting to the terminal, comments are suppressed and printed as the string **COMMENT**. The value of **COMMENT**FLG determines the action.

> ****COMMENT**FLG**   [Variable]
>
> > If **COMMENT**FLG is NIL, comments are printed. Otherwise, the value of **COMMENT**FLG is printed. Initially " **COMMENT** ".

> (**COMMENT1** L —)   [Function]
>
> > Prints the comment L. COMMENT1 is a separate function to permit the user to write prettyprint macros that use the regular comment printer. For example, to cause comments to be printed at a larger than normal linelength, one could put an entry for * on PRETTYPRINTMACROS:
> >
> > ```
> > (* LAMBDA (X) (RESETFORM (LINELENGTH 100)
> > (COMMENT1 X)))
> > ```
> >
> > This macro resets the line length, prints the comment, and then restores the line length.
> >
> > COMMENT1 expects to be called from within the environment established by PRINTDEF, so ordinarily the user should call it only from within prettyprint macros.

Comment Pointers

For a well-commented collection of programs, the list structure, atom, and print name storage required to represent the comments in core can be significant. If the comments already appear on a file and are not needed for editing, a significant savings in storage can be achieved by simply leaving the text of the comment on the file when the file is loaded, and instead retaining in core only a pointer to the comment. When this feature is enabled, * is defined as a read macro (see Chapter 25) in

`FILERDTBL` which, instead of reading in the entire text of the comment, constructs an expression containing

- The name of the file in which the text of the comment is contained

- The address of the first character of the comment

- The number of characters in the comment

- A flag indicating whether the comment appeared at the right hand margin or centered on the page

For output purposes, * is defined on `PRETTYPRINTMACROS` (see the Prettyprint Control Functions section below) so that it prints the comments represented by such pointers by simply copying the corresponding characters from one file to another, or to the terminal. Normal comments are processed the same as before, and can be intermixed freely with comment pointers.

The comment pointer feature is controlled by the function `NORMALCOMMENTS`.

> (**NORMALCOMMENTS** *FLG*) [Function]
>
> > If *FLG* is `NIL`, the comment pointer feature is enabled. If *FLG* is `T`, the comment pointer feature is disabled (the default).
> >
> > `NORMALCOMMENTS` can be changed as often as desired. Thus, some files can be loaded normally, and others with their comments converted to comment pointers.

For convenience of editing selected comments, an edit macro, `GET*`, is included, which loads in the text of the corresponding comment. The editor's `PP*` command, in contrast, prints the comment without reading it by simply copying the corresponding characters to the terminal. `GET*` is defined in terms of `GETCOMMENT`:

> (**GETCOMMENT** *X DESTFL —*) [Function]
>
> > If *X* is a comment pointer, replaces *X* with the actual text of the comment, which it reads from its file. Returns *X* in all cases. If *DESTFL* is non-`NIL`, it is the name of an open file, to which `GETCOMMENT` copies the comment; in this case, *X* remains a comment pointer, but it has been changed to point to the new file (unless `NORMALCOMMENTS` has been set to `DONTUPDATE`).

> (**PRINTCOMMENT** *X*) [Function]
>
> > Defined as the prettyprint macro for *: copies the comment to the primary output file by using `GETCOMMENT`.

> (**READCOMMENT** *FL RDTBL LST*) [Function]
>
> > Defined as the read macro for * in `FILERDTBL`: if `NORMALCOMMENTSFLG` is `NIL`, it constructs a comment pointer, unless it believes the expression beginning with * is not actually a comment, e.g., if the next atom is "." or `E`.

Note that a certain amount of care is required in using the comment pointer feature. Since the text of the comment resides on the file pointed to by the comment pointer, that file must remain in existence as long as the comment is needed. `GETCOMMENT` helps out by changing the comment pointer to always point at the most recent file that the comment lives on. However, if the user has been performing repeated `MAKEFILE`'s (see Chapter 17) in which differing functions have changed at each invocation of `MAKEFILE`, it is possible for the comment pointers in memory to be pointing at several versions of the same file, since a comment pointer is only updated when the function it lives in is prettyprinted, not when the function has been copied verbatim to the new file. This can be a problem for file systems that have a built-in limit on the number of versions of a given file that will be made before old versions are expunged. In such a case, the user should set the version retention count of any directories involved to be infinite. `GETCOMMENT` prints an error message if the file that the comment pointer points at has disappeared.

Similarly, one should be cognizant of comment pointers in sysouts, and be sure to retain any files thus pointed to.

When using comment pointers, the user should also not set `PRETTYFLG` to `NIL` or call `MAKEFILE` with option `FAST`, since this will prevent functions from being prettyprinted, and hence not get the text of the comment copied into the new file.

If the user changes the value of `COMMENTFLG` but still wishes to use the comment pointer feature, the new `COMMENTFLG` should be given the same read-macro definition in `FILERDTBL` as `*` has, and the same entry be put on `PRETTYPRINTMACROS`. For example, if `COMMENTFLG` is reset to be `";"`, then `(SETSYNTAX '; '* FILERDTBL)` should be performed, and `(; . PRINTCOMMENT)` added to `PRETTYPRINTMACROS`.

Converting Comments to Lower Case

This section is for users using terminals without lower case, who nevertheless would like their comments to be converted to lower case for more readable listings. If the second atom in a comment is `%%`, the text of the comment is converted to lower case so that it looks like English instead of Lisp. Note that comments are converted only when they are actually written to a file by `PRETTYPRINT`.

The algorithm for conversion to lower case is the following: If the first character in an atom is `^`, do not change the atom (but remove the `^`). If the first character is `%`, convert the atom to lower case. Note that the user must type `%%` as `%` is the escape character. If the atom (minus any trailing punctuation marks) is an Interlisp word (i.e., is a bound or free variable for the function containing the comment, or has a top level value, or is a defined function, or has a non-`NIL` property list), do not change it. Otherwise, convert the atom to lower case. Conversion only affects the upper case alphabet, i.e., atoms already converted to lower case are not changed if the comment is converted again. When converting, the first character in the comment and the first character following each period are left capitalized. After conversion, the comment is physically modified to be the lower case text minus the `%%` flag, so that conversion is thus only performed once (unless the user edits the comment inserting additional upper case text and another `%%` flag).

LCASELST [Variable]

Words on `LCASELST` will always be converted to lower case. `LCASELST` is initialized to contain words which are Interlisp functions but also appear frequently in comments as English words (`AND`, `EVERY`, `GET`, `GO`, `LAST`, `LENGTH`, `LIST`, etc.). Therefore, if one wished to type a comment including the lisp fuction `GO`, it would be necessary to type ↑`GO` in order that it might be left in upper case.

UCASELST [Variable]

Words on `UCASELST` (that do not appear on `LCASELST`) will be left in upper case. `UCASELST` is initialized to `NIL`.

ABBREVLST [Variable]

`ABBREVLST` is used to distinguish between abbreviations and words that end in periods. Normally, words that end in periods and occur more than halfway to the right margin cause carriage-returns. Furthermore, during conversion to lowercase, words ending in periods, except for those on `ABBREVLST`, cause the first character in the next word to be capitalized. `ABBREVLST` is initialized to the upper and lower case forms of `ETC.`, `I.E.`, and `E.G.`.

Special Prettyprint Controls

PRETTYTABFLG [Variable]

In order to save space on files, tabs are used instead of spaces for the inital spaces on each line, assuming that each tab corresponds to 8 spaces. This results in a reduction of file size by about 30%. Tabs are not used if `PRETTYTABFLG` is set to `NIL` (initially `T`).

#RPARS [Variable]

Controls the number of right parentheses necessary for square bracketing to occur. If `#RPARS = NIL`, no brackets are used. `#RPARS` is initialized to 4.

FIRSTCOL [Variable]

The starting column for comments. Comments run between `FIRSTCOL` and the line length set by `LINELENGTH` (see Chapter 25). If a word in a comment ends with a `"."` and is not on the list `ABBREVLST`, and the position is greater than halfway between `FIRSTCOL` and `LINELENGTH`, the next word in the comment begins on a new line. Also, if a list is encountered in a comment, and the position is greater than halfway, the list begins on a new line.

PRETTYLCOM [Variable]

If a comment has more than `PRETTYLCOM` elements (using `COUNT`), it is printed starting at column 10, instead of `FIRSTCOL`. Comments are also printed starting at column 10 if their second element is also a `*`, i.e., comments of the form `( * * -- )`.

#CAREFULCOLUMNS [Variable]

In the interests of efficiency, PRETTYPRINT approximates the number of characters in each atom, rather than calling NCHARS, when computing how much will fit on a line. This procedure works satisfactorily in most cases. However, users with unusually long atoms in their programs, e.g., such as produced by CLISPIFY, may occasionlly encounter some glitches in the output produced by PRETTYPRINT. The value of #CAREFULCOLUMNS tells PRETTYPRINT how many columns (counting from the right hand margin) in which to actually compute NCHARS instead of approximating. Setting #CAREFULCOLUMNS to 20 or 30 will eliminate the glitches, although it will slow down PRETTYPRINT slightly. #CAREFULCOLUMNS is initially 0.

(**WIDEPAPER** *FLG*) [Function]

(WIDEPAPER T) sets FILELINELENGTH (see Chapter 25), FIRSTCOL, and PRETTYLCOM to large values appropriate for pretty printing files to be listed on wide paper. (WIDEPAPER) restores these parameters to their initial values. WIDEPAPER returns the previous setting of *FLG*.

PRETTYFLG [Variable]

If PRETTYFLG is NIL, PRINTDEF uses PRIN2 instead of prettyprinting. This is useful for producing a fast symbolic dump (see the FAST option of MAKEFILE in Chapter 17). Note that the file loads the same as if it were prettyprinted. PRETTYFLG is initially set to T. PRETTYFLG should not be set to NIL if comment pointers are being used.

CLISPIFYPRETTYFLG [Variable]

Used to inform PRETTYPRINT to call CLISPIFY on selected function definitions before printing them (see Chapter 21).

PRETTYPRINTMACROS [Variable]

An association-list that enables the user to control the formatting of selected expressions. CAR of each expression being PRETTYPRINTed is looked up on PRETTYPRINTMACROS, and if found, CDR of the corresponding entry is applied to the expression. If the result of this application is NIL, PRETTYPRINT ignores the expression; i.e., it prints nothing, assuming that the prettyprintmacro has done any desired printing. If the result of applying the prettyprint macro is non-NIL, the result is prettyprinted in the normal fashion. This gives the user the option of computing some other expression to be prettyprinted in its place.

Note: "prettyprinted in the normal fashion" includes processing prettyprint macros, unless the prettyprint macro returns a structure EQ to the one it was handed, in which case the potential recursion is broken.

PRETTYPRINTYPEMACROS [Variable]

A list of elements of the form (TYPENAME . FN). For types other than lists and atoms, the type name of each datum to be prettyprinted is looked up on

PRETTYPRINTYPEMACROS, and if found, the corresponding function is applied to the datum about to be printed, instead of simply printing it with PRIN2.

PRETTYEQUIVLST [Variable]

An association-list that tells PRETTYPRINT to treat a CAR-of-form the same as some other CAR-of-form. For example, if (QLAMBDA . LAMBDA) appears on PRETTYEQUIVLST, then expressions beginning with QLAMBDA are prettyprinted the same as LAMBDAs. Currently, PRETTYEQUIVLST only allows (i.e., supports in an interesting way) equivalences to forms that PRETTYPRINT internally handles. Equivalence to forms for which the user has specified a prettyprint macro should be made by adding further entries to PRETTYPRINTMACROS

CHANGECHAR [Variable]

If non-NIL, and PRETTYPRINT is printing to a file or display terminal, PRETTYPRINT prints CHANGECHAR in the right hand margin while printing those expressions marked by the editor as having been changed (see Chapter 16). CHANGECHAR is initially |.

26. GRAPHICS OUTPUT OPERATIONS

Streams are used as the basis for all I/O operations. Files are implemented as streams that can support character printing and reading operations, and file pointer manipulation. An image stream is a type of stream that also provides an interface for graphical operations. All of the operations that can applied to streams can be applied to image streams. For example, an image stream can be passed as the argument to PRINT, to print something on an image stream. In addition, special functions are provided to draw lines and curves and perform other graphical operations. Calling these functions on a stream that is not an image stream will generate an error.

Primitive Graphics Concepts

The Interlisp-D graphics system is based on manipulating bitmaps (rectangular arrays of pixels), positions, regions, and textures. These objects are used by all of the graphics functions.

Positions

A position denotes a point in an X,Y coordinate system. A POSITION is an instance of a record with fields XCOORD and YCOORD and is manipulated with the standard record package facilities. For example, (create POSITION XCOORD ← 10 YCOORD ← 20) creates a position representing the point (10,20).

> (**POSITIONP** *X*)   [Function]
>
> Returns *X* if *X* is a position; NIL otherwise.

Regions

A Region denotes a rectangular area in a coordinate system. Regions are characterized by the coordinates of their bottom left corner and their width and height. A REGION is a record with fields LEFT, BOTTOM, WIDTH, and HEIGHT. It can be manipulated with the standard record package facilities. There are access functions for the REGION record that return the TOP and RIGHT of the region.

The following functions are provided for manipulating regions:

> (**CREATEREGION** *LEFT BOTTOM WIDTH HEIGHT*)   [Function]
>
> Returns an instance of the REGION record which has *LEFT*, *BOTTOM*, *WIDTH* and *HEIGHT* as respectively its *LEFT*, *BOTTOM*, *WIDTH*, and *HEIGHT* fields.

Example: (CREATEREGION 10 -20 100 200) will create a region that denotes a rectangle whose width is 100, whose height is 200, and whose lower left corner is at the position (10,-20).

(**REGIONP** *X*) [Function]

Returns *X* if *X* is a region, NIL otherwise.

(**INTERSECTREGIONS** *REGION$_1$ REGION$_2$ ... REGION$_n$*) [NoSpread Function]

Returns a region which is the intersection of a number of regions. Returns NIL if the intersection is empty.

(**UNIONREGIONS** *REGION$_1$ REGION$_2$ ... REGION$_n$*) [NoSpread Function]

Returns a region which is the union of a number of regions, i.e. the smallest region that contains all of them. Returns NIL if there are no regions given.

(**REGIONSINTERSECTP** *REGION$_1$ REGION$_2$*) [Function]

Returns T if *REGION$_1$* intersects *REGION$_2$*. Returns NIL if they do not intersect.

(**SUBREGIONP** *LARGEREGION SMALLREGION*) [Function]

Returns T if *SMALLREGION* is a subregion (is equal to or entirely contained in) *LARGEREGION*; otherwise returns NIL.

(**EXTENDREGION** *REGION INCLUDEREGION*) [Function]

Changes (destructively modifies) the region *REGION* so that it includes the region *INCLUDEREGION*. It returns *REGION*.

(**MAKEWITHINREGION** *REGION LIMITREGION*) [Function]

Changes (destructively modifies) the left and bottom of the region *REGION* so that it is within the region *LIMITREGION*, if possible. If the dimension of *REGION* are larger than *LIMITREGION*, *REGION* is moved to the lower left of *LIMITREGION*. If *LIMITREGION* is NIL, the value of the variable *WHOLEDISPLAY* (the screen region) is used. MAKEWITHINREGION returns the modified *REGION*.

(**INSIDEP** *REGION POSORX Y*) [Function]

If *POSORX* and *Y* are numbers, it returns T if the point (*POSORX,Y*) is inside of *REGION*. If *POSORX* is a *POSITION*, it returns T if *POSORX* is inside of *REGION*. If *REGION* is a WINDOW, the window's interior region in window coordinates is used. Otherwise, it returns NIL.

Bitmaps

The display primitives manipulate graphical images in the form of bitmaps. A bitmap is a rectangular array of "pixels," each of which is an integer representing the color of one point in the bitmap image. A bitmap is created with a specific number of bits allocated for each pixel. Most bitmaps used for the display screen use one bit per pixel, so that at most two colors can be represented. If a pixel is 0, the corresponding location on the image is white. If a pixel is 1, its location is black. This interpretation can be changed for the display screen with the function VIDEOCOLOR. Bitmaps with more than one bit per pixel are used to represent color or grey scale images. Bitmaps use a positive integer coordinate system with the lower left corner pixel at coordinate (0,0). Bitmaps are represented as instances of the datatype BITMAP. Bitmaps can be saved on files with the VARS file package command.

(**BITMAPCREATE** *WIDTH HEIGHT BITSPERPIXEL*) [Function]

> Creates and returns a new bitmap which is *WIDTH* pixels wide by *HEIGHT* pixels high, with *BITSPERPIXEL* bits per pixel. If BITSPERPIXEL is NIL, it defaults to 1.

(**BITMAPP** *X*) [Function]

> Returns *X* if *X* is a bitmap, NIL otherwise.

(**BITMAPWIDTH** *BITMAP*) [Function]

> Returns the width of *BITMAP* in pixels.

(**BITMAPHEIGHT** *BITMAP*) [Function]

> Returns the height of *BITMAP* in pixels.

(**BITSPERPIXEL** *BITMAP*) [Function]

> Returns the number of bits per pixel of *BITMAP*.

(**BITMAPBIT** *BITMAP X Y NEWVALUE*) [Function]

> If *NEWVALUE* is between 0 and the maximum value for a pixel in *BITMAP*, the pixel (*X,Y*) is changed to *NEWVALUE* and the old value is returned. If *NEWVALUE* is NIL, *BITMAP* is not changed but the value of the pixel is returned. If *NEWVALUE* is anything else, an error is generated. If (*X,Y*) is outside the limits of *BITMAP*, 0 is returned and no pixels are changed. *BITMAP* can also be a window or display stream. Note: non-window image streams are "write-only"; the *NEWVALUE* argument must be non-NIL.

(**BITMAPCOPY** *BITMAP*) [Function]

> Returns a new bitmap which is a copy of *BITMAP* (same dimensions, bits per pixel, and contents).

(**EXPANDBITMAP** *BITMAP WIDTHFACTOR HEIGHTFACTOR*) [Function]

> Returns a new bitmap that is *WIDTHFACTOR* times as wide as *BITMAP* a

nd *HEIGHTFACTOR* times as high. Each pixel of $BITMAP$ is copied into a *WIDTHFACTOR* times *HEIGHTFACTOR* block of pixels. If NIL, *WIDTHFACTOR* defaults to 4, *HEIGHTFACTOR* to 1.

(**ROTATEBITMAP** *BITMAP*)   [Function]

Given an m-high by n-wide bitmap, this function returns an n-high by m-wide bitmap. The returned bitmap is the image of the original bitmap, rotated 90 degrees clockwise.

(**SHRINKBITMAP** *BITMAP WIDTHFACTOR HEIGHTFACTOR DESTINATIONBITMAP*)
 [Function]

Returns a copy of *BITMAP* that has been shrunken by *WIDTHFACTOR* and *HEIGHTFACTOR* in the width and height, respectively. If NIL, *WIDTHFACTOR* defaults to 4, *HEIGHTFACTOR* to 1. If *DESTINATIONBITMAP* is not provided, a bitmap that is 1/*WIDTHFACTOR* by 1/*HEIGHTFACTOR* the size of *BITMAP* is created and returned. *WIDTHFACTOR* and *HEIGHTFACTOR* must be positive integers.

(**PRINTBITMAP** *BITMAP FILE*)   [Function]

Prints the bitmap *BITMAP* on the file *FILE* in a format that can be read back in by READBITMAP.

(**READBITMAP** *FILE*)   [Function]

Creates a bitmap by reading an expression (written by PRINTBITMAP) from the file *FILE*.

(**EDITBM** *BMSPEC*)   [Function]

EDITBM provides an easy-to-use interactive editing facility for various types of bitmaps. If *BMSPEC* is a bitmap, it is edited. If *BMSPEC* is an atom whose value is a bitmap, its value is edited. If *BMSPEC* is NIL, EDITBM asks for dimensions and creates a bitmap. If *BMSPEC* is a region, that portion of the screen bitmap is used. If *BMSPEC* is a window, it is brought to the top and its contents edited.

EDITBM sets up the bitmap being edited in an editing window. The editing window has two major areas: a gridded edit area in the lower part of the window and a display area in the upper left part. In the edit area, the left button will add points, the middle button will erase points. The right button provides access to the normal window commands to reposition and reshape the window. The actual size bitmap is shown in the display area. For example, the following is a picture of the bitmap editing window editing a eight-high by eighteen-wide bitmap:
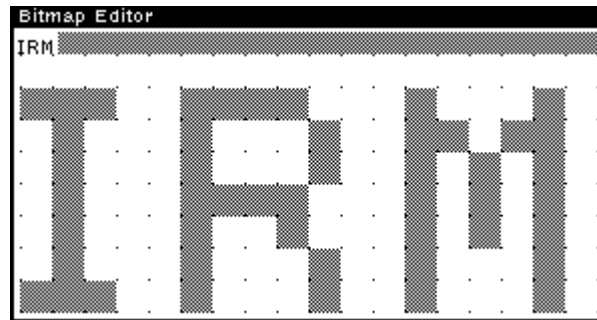
If the bitmap is too large to fit in the edit area, only a portion will be editable. This portion can be changed by scrolling both up and down in the left margin and left and right in the bottom margin. Pressing the middle button while in the display area will bring up a menu that allows global placement of the portion of the bitmap being edited. To allow more of the bitmap to be editing at once, the window can be reshaped to make it larger or the GridSize← command described below can be used to reduce the size of a bit in the edit area.

The bitmap editing window can be reshaped to provide more or less room for editing. When this happens, the space allocated to the editing area will be changed to fit in the new region.

Whenever the left or middle button is down and the cursor is not in the edit area, the section of the display of the bitmap that is currently in the edit area is complemented. Pressing the left button while not in the edit region will put the lower left 16 x 16 section of the bitmap into the cursor for as long as the left button is held down.

Pressing the middle button while not in either the edit area or the display area (i.e., while in the grey area in the upper right or in the title) will bring up a command menu.
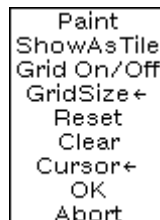
There are commands to stop editing, to restore the bitmap to its initial state and to clear the bitmap. Holding the middle button down over a command will result in an explanatory message being printed in the prompt window. The commands are described below:

Paint Puts the current bitmap into a window and call the window PAINT command on it. The PAINT command implements drawing with various brush sizes and shapes but only on an actual sized bitmap. The PAINT mode is left by pressing the RIGHT button and selecting the QUIT command from

the menu. At this point, you will be given a choice of whether or not the changes you made while in PAINT mode should be made to the current bitmap.

ShowAsTile Tesselates the current bitmap in the upper part of the window. This is useful for determining how a bitmap will look if it were made the display background (using the function CHANGEBACKGROUND). Note: The tiled display will not automatically change as the bitmap changes; to update it, use the ShowAsTile command again.

Grid,On/Off Turns the editing grid display on or off.

GridSize← Allows specification of the size of the editing grid. Another menu will appear giving a choice of several sizes. If one is selected, the editing portion of the bitmap editor will be redrawn using the selected grid size, allowing more or less of the bitmap to be edited without scrolling. The original size is chosen hueristically and is typically about 8. It is particularly useful when editing large bitmaps to set the edit grid size smaller than the original.

Reset Sets all or part of the bitmap to the contents it had when EDITBM was called. Another menu will appear giving a choice between resetting the entire bitmap or just the portion that is in the edit area. The second menu also acts as a confirmation, since not selecting one of the choices on this menu results in no action being taken.

Clear Sets all or part of the bitmap to 0. As with the Reset command, another menu gives a choice between clearing the entire bitmap or just the portion that is in the edit area.

Cursor← Sets the cursor to the lower left part of the bitmap. This prompts the user to specify the cursor "hot spot" by clicking in the lower left corner of the grid.

OK Copies the changed image into the original bitmap, stops the bitmap editor and closes the edit windows. The changes the bitmap editor makes during the interaction occur on a copy of the original bitmap. Unless the bitmap editor is exited via OK, no changes are made in the original.

Stop Stops the bitmap editor without making any changes to the original bitmap.

Textures

A Texture denotes a pattern of gray which can be used to (conceptually) tessellate the plane to form an infinite sheet of gray. It is currently either a 4 by 4 pattern or a 16 by N (N <= 16) pattern. Textures are created from bitmaps using the following function:

(**CREATETEXTUREFROMBITMAP** *BITMAP*)   [Function]

Returns a texture object that will produce the texture of *BITMAP*. If *BITMAP* is too large, its lower left portion is used. If *BITMAP* is too small, it is repeated to fill out the texture.

(**TEXTUREP** *OBJECT*) [Function]

> Returns *OBJECT* if it is a texture; NIL otherwise.

The functions which accept textures (TEXTUREP, BITBLT, DSPTEXTURE, etc.) also accept bitmaps up to 16 bits wide by 16 bits high as textures. When a region is being filled with a bitmap texture, the texture is treated as if it were 16 bits wide (if less, the rest is filled with white space).

The common textures white and black are available as system constants WHITESHADE and BLACKSHADE. The global variable GRAYSHADE is used by many system facilities as a background gray shade and can be set by the user.

(**EDITSHADE** *SHADE*) [Function]

> Opens a window that allows the user to edit textures. Textures can be either small (4 by 4) patterns or large (16 by 16). In the edit area, the left button adds bits to the shade and the middle button erases bits from the shade. The top part of the window is painted with the current texture whenever all mouse keys are released. Thus it is possible to directly compare two textures that differ by more than one pixel by holding a mouse key down until all changes are made. When the "quit" button is selected, the texture being edited is returned.

> If *SHADE* is a texture object, EDITSHADE starts with it. If *SHADE* is T, it starts with a large (16 by 16) white texture. Otherwise, it starts with WHITESHADE.

The following is a picture of the texture editor, editing a large (16 by 16) pattern:
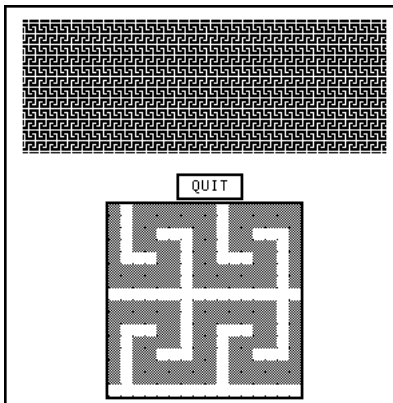
Opening Image Streams

An image stream is an output stream which "knows" how to process graphic commands to a graphics output device. Besides accepting the normal character-output functions (PRINT, etc.), an image

stream can also be passed as an argument to functions to draw curves, to print characters in multiple fonts, and other graphics operations.

Each image stream has an "image stream type," a litatom that specifies the type of graphic output device that the image stream is processing graphics commands for. Currently, the built-in image stream types are DISPLAY (for the display screen), INTERPRESS (for Interpress format printers), and PRESS (for Press format printers). There are also library packages **available that define image stream types for the IRIS display, 4045 printer, FX-80 printer, C150 printer, etc.**

Image streams to the display (display streams) interpret graphics commands by immediately executing the appropriate operations to cause the desired image to appear on the display screen. Image streams for hardcopy devices such as Interpress printers interpret the graphic commands by saving information in a file, which can later be sent to the printer.

Note: Not all graphics operations can be properly executed for all image stream types. For example, BITBLT may not be supported to all printers. This functionality is still being developed, but even in the long run some operations may be beyond the physical or logical capabilities of some devices or image file formats. In these cases, the stream will approximate the specified image as best it can.

(**OPENIMAGESTREAM** *FILE IMAGETYPE OPTIONS*)   [Function]

> Opens and returns an image stream of type *IMAGETYPE* on a destination specified by *FILE*. If *FILE* is a file name on a normal file storage device, the image stream will store graphics commands on the specified file, which can be transmitted to a printer by explicit calls to LISTFILES and SEND.FILE.TO.PRINTER. If *IMAGETYPE* is DISPLAY, then the user is prompted for a window to open. *FILE* in this case will be used as the title of the window.

> If *FILE* is a file name on the LPT device, this indicates that the graphics commands should be stored in a temporary file, and automatically sent to the printer when the image stream is closed by CLOSEF. *FILE* = NIL is equivalent to *FILE* = {LPT}. File names on the LPT device are of the form {LPT}PRINTERNAME.TYPE, where PRINTERNAME, TYPE, or both may be omitted.   PRINTERNAME is the name of the particular printer to which the file will be transmitted on closing; it defaults to the first printer on DEFAULTPRINTINGHOST that can print *IMAGETYPE* files. The TYPE extension supplies the value of *IMAGETYPE* when it is defaulted (see below). OPENIMAGESTREAM will generate an error if the specified printer does not accept the kind of file specified by *IMAGETYPE*.

> If *IMAGETYPE* is NIL, the image type is inferred from the extension field of *FILE* and the EXTENSIONS properties in the list PRINTFILETYPES. Thus, the extensions IP, IPR, and INTERPRESS indicate Interpress format, and the extension PRESS indicates Press format. If *FILE* is a printer file with no extension (of the form {LPT}PRINTERNAME), then *IMAGETYPE* will be the type that the indicated printer can print. If *FILE* has no extension but is not on the printer device {LPT}, then *IMAGETYPE* will default to the type accepted by the first printer on DEFAULTPRINTINGHOST.

OPTIONS is a list in property list format, (PROP1 VAL1 PROP2 VAL2 —), used to specify certain attributes of the image stream; not all attributes are meaningful or interpreted by all types of image streams. Acceptable properties are:

REGION   Value is the region on the page (in stream scale units, 0,0 being the lower-left corner of the page) that text will fill up. It establishes the initial values for DSPLEFTMARGIN, DSPRIGHTMARGIN, DSPBOTTOMMARGIN (the point at which carriage returns cause page advancement) and DSPTOPMARGIN (where the stream is positioned at the beginning of a new page).

If this property is not given, the value of the variable DEFAULTPAGEREGION, is used.

FONTS   Value is a list of fonts that are expected to be used in the image stream. Some image streams (e.g. Interpress) are more efficient if the expected fonts are specified in advance, but this is not necessary. The first font in this list will be the initial font of the stream, otherwise the default font for that image stream type will be used.

HEADING Value is the heading to be placed automatically on each page. NIL means no heading.

Examples: Suppose that Tremor: is an Interpress printer, Quake is a Press printer, and DEFAULTPRINTINGHOST is (Tremor: Quake):

(OPENIMAGESTREAM)   returns an Interpress image stream on printer Tremor:.

(OPENIMAGESTREAM NIL 'PRESS) returns a Press stream on Quake.

(OPENIMAGESTREAM '{LPT}.INTERPRESS)   returns an Interpress stream on Tremor:.

(OPENIMAGESTREAM '{CORE}FOO.PRESS) returns a Press stream on the file {CORE}FOO.PRESS.

(**IMAGESTREAMP** *X IMAGETYPE*)   [NoSpread Function]

Returns *X* (possibly coerced to a stream) if it is an output image stream of type *IMAGETYPE* (or of any type if *IMAGETYPE* = NIL), otherwise NIL.

(**IMAGESTREAMTYPE** *STREAM*)   [Function]

Returns the image stream type of *STREAM*.

(**IMAGESTREAMTYPEP** *STREAM TYPE*)   [Function]

Returns T if *STREAM* is an image stream of type *TYPE*.

Accessing Image Stream Fields

The following functions manipulate the fields of an image stream. These functions return the old value (the one being replaced). A value of NIL for the new value will return the current setting without changing it. These functions do not change any of the bits drawn on the image stream; they just affect future operations done on the image stream.

(**DSPCLIPPINGREGION** *REGION STREAM*)   [Function]

> The clipping region is a region that limits the extent of characters printed and lines drawn (in the image stream's coordinate system). Initially set so that no clipping occurs.

> Warning: For display streams, the window system maintains the clipping region during window operations. Users should be very careful about changing this field.

(**DSPFONT** *FONT STREAM*)   [Function]

> The font field specifies the font used when printing characters to the image stream.

>> Note: DSPFONT determines its new font descriptor from *FONT* by the same coercion rules that FONTPROP and FONTCREATE use , with one additional possibility: If *FONT* is a list of the form (PROP$_1$ VAL$_1$ PROP$_2$ VAL$_2$ ...) where PROP$_1$ is acceptable as a font-property to FONTCOPY, then the new font is obtained by (FONTCOPY (DSPFONT NIL STREAM) PROP$_1$ VAL$_1$ PROP$_2$ VAL$_2$ ...). For example, (DSPFONT '(SIZE 12) STREAM) would change the font to the 12 point version of the current font, leaving all other font properties the same.

(**DSPTOPMARGIN** *YPOSITION STREAM*)   [Function]

> The top margin is an integer that is the Y position after a new page (in the image stream's coordinate system). This function has no effect on windows.

(**DSPBOTTOMMARGIN** *YPOSITION STREAM*)   [Function]

> The bottom margin is an integer that is the minimum Y position that characters will be printed by PRIN1 (in the image stream's coordinate system). This function has no effect on windows.

(**DSPLEFTMARGIN** *XPOSITION STREAM*)   [Function]

> The left margin is an integer that is the X position after an end-of-line (in the image stream's coordinate system). Initially the left edge of the clipping region.

(**DSPRIGHTMARGIN** *XPOSITION STREAM*)   [Function]

> The right margin is an integer that is the maximum X position that characters will be printed by PRIN1 (in the image stream's coordinate system). This is initially the position of the right edge of the window or page.

The line length of a window or image stream (as returned by LINELENGTH) is computed by dividing the distance between the left and right margins by the width of an uppercase "A" in the current font. The line length is changed whenever the font, left margin, or right margin are changed or whenever the window is reshaped.

(**DSPOPERATION** *OPERATION STREAM*)   [Function]

> The operation is the default BITBLT operation used when printing or drawing on the image stream. One of REPLACE, PAINT, INVERT, or ERASE. Initially REPLACE. This is a meaningless operation for most printers which support the model that once dots are deposited on a page they cannot be removed.

(**DSPLINEFEED** *DELTAY STREAM*)   [Function]

> The linefeed is an integer that specifies the *Y* increment for each linefeed, normally negative. Initially minus the height of the initial font.

(**DSPCLEOL** *DSPSTREAM XPOS YPOS HEIGHT*)   [Function]

> "Clear to end of line". Clears a region from (*XPOS*,*YPOS*) to the right margin of the display, with a height of *HEIGHT*. If *XPOS* and *YPOS* are NIL, clears the remainder of the current display line, using the height of the current font.

(**DSPRUBOUTCHAR** *DSPSTREAM CHAR X Y TTBL*)   [Function]

> Backs up over character code *CHAR* in the *DSPSTREAM*, erasing it. If *X*, *Y* are supplied, the rubbing out starts from the position specified. **DSPRUBOUTCHAR** assumes *CHAR* was printed with the terminal table *TTBL*, so it knows to handle control characters, etc. *TTBL* defaults to the primary terminal table.

(**DSPSCALE** *SCALE STREAM*)   [Function]

> Returns the scale of the image stream *STREAM*, a number indicating how many units in the streams coordinate system correspond to one printer's point (1/72 of an inch). For example, DSPSCALE returns 1 for display streams, and 35.27778 for Interpress and Press streams (the number of micas per printer's point). In order to be device-independent, user graphics programs must either not specify position values absolutely, or must multiply absolute point quantities by the DSPSCALE of the destination stream. For example, to set the left margin of the Interpress stream XX to one inch, do

```
(DSPLEFTMARGIN (TIMES 72 (DSPSCALE NIL XX)) XX)
```

The SCALE argument to DSPSCALE is currently ignored. In a future release it will enable the scale of the stream to be changed under user control, so that the necessary multiplication will be done internal to the image stream interface. In this case, it would be possible to set the left margin of the Interpress stream XX to one inch by doing

```
(DSPSCALE 1 XX)
(DSPLEFTMARGIN 72 XX)
```

(**DSPSPACEFACTOR** *FACTOR STREAM*)   [Function]

>The space factor is the amount by which to multiply the natural width of all following space characters on *STREAM*; this can be used for the justification of text. The default value is 1. For example, if the natural width of a space in *STREAM*'s current font is 12 units, and the space factor is set to two, spaces appear 24 units wide. The values returned by STRINGWIDTH and CHARWIDTH are also affected.

The following two functions only have meaning for image streams that can display color:

(**DSPCOLOR** *COLOR STREAM*)   [Function]

>Sets the default foreground color of *STREAM*. Returns the previous foreground color. If *COLOR* is NIL, it returns the current foreground color without changing anything. The default color is white

(**DSPBACKCOLOR** *COLOR STREAM*)   [Function]

>Sets the background color of *STREAM*. Returns the previous background color. If *COLOR* is NIL, it returns the current background color without changing anything. The default background color is black.

Current Position of an Image Stream

Each image stream has a "current position," which is a position (in the image stream's coordinate system) where the next printing operation will start from. The functions which print characters or draw on an image stream update these values appropriately. The following functions are used to explicitly access the current position of an image stream:

(**DSPXPOSITION** *XPOSITION STREAM*)   [Function]

>Returns the X coordinate of the current position of *STREAM*. If *XPOSITION* is non-NIL, the X coordinate is set to it (without changing the Y coordinate).

(**DSPYPOSITION** *YPOSITION STREAM*)   [Function]

>Returns the Y coordinate of the current position of *STREAM*. If *YPOSITION* is non-NIL, the Y coordinate is set to it (without changing the X coordinate).

(**MOVETO** *X Y STREAM*)   [Function]

>Changes the current position of *STREAM* to the point (*X,Y*).

(**RELMOVETO** *DX DY STREAM*)   [Function]

>Changes the current position to the point (*DX,DY*) coordinates away from current position of *STREAM*.

(**MOVETOUPPERLEFT** *STREAM REGION*)   [Function]

> Moves the current position to the beginning position of the top line of text. If *REGION* is non-NIL, it must be a *REGION* and the X position is changed to the left edge of *REGION* and the Y position changed to the top of *REGION* less the font ascent of *STREAM*. If *REGION* is NIL, the X coordinate is changed to the left margin of *STREAM* and the Y coordinate is changed to the top of the clipping region of *STREAM* less the font ascent of *STREAM*.

Moving Bits Between Bitmaps With BITBLT

BITBLT is the primitive function for moving bits from one bitmap to another, or from a bitmap to an image stream.

(**BITBLT** *SOURCE SOURCELEFT SOURCEBOTTOM DESTINATION DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT SOURCETYPE OPERATION TEXTURE CLIPPINGREGION*)   [Function]

> Transfers a rectangular array of bits from *SOURCE* to *DESTINATION*. *SOURCE* can be a bitmap, or a display stream or window, in which case its associated bitmap is used. *DESTINATION* can be a bitmap or an arbitrary image stream.
>
> *WIDTH* and *HEIGHT* define a pair of rectangles, one in each of the *SOURCE* and *DESTINATION* whose left, bottom corners are at, respectively, (*SOURCELEFT*, *SOURCEBOTTOM*) and (*DESTINATIONLEFT*, *DESTINATIONBOTTOM*). If these rectangles overlap the boundaries of either source or destination they are both reduced in size (without translation) so that they fit within their respective boundaries. If *CLIPPINGREGION* is non-NIL it should be a *REGION* and is interpreted as a clipping region within *DESTINATION*; clipping to this region may further reduce the defining rectangles. These (possibly reduced) rectangles define the source and destination rectangles for BITBLT.
>
> The mode of *transferring* bits is defined by *SOURCETYPE* and *OPERATION*. *SOURCETYPE* and *OPERATION* specify whether the source bits should come from *SOURCE* or *TEXTURE*, and how these bits are combined with those of *DESTINATION*. *SOURCETYPE* and *OPERATION* are described further below.
>
> *TEXTURE* is a texture. BITBLT aligns the texture so that the upper-left pixel of the texture coincides with the upper-left pixel of the destination bitmap.
>
> *SOURCELEFT*, *SOURCEBOTTOM*, *DESTINATIONLEFT*, and *DESTINATIONBOTTOM* default to 0. *WIDTH* and *HEIGHT* default to the width and height of the *SOURCE*. *TEXTURE* defaults to white. *SOURCETYPE* defaults to INPUT. *OPERATION* defaults to REPLACE. If *CLIPPINGREGION* is not provided, no additional clipping is done. BITBLT returns T if any bits were moved; NIL otherwise.
>
> Note: If *SOURCE* or *DESTINATION* is a window or image stream, the remaining arguments are interpreted as values in the coordinate system of the window or image

stream and the operation of `BITBLT` is translated and clipped accordingly. Also, if a window or image stream is used as the destination to `BITBLT`, its clipping region further limits the region involved.

SOURCETYPE specifies whether the source bits should come from the bitmap *SOURCE*, or from the texture *TEXTURE*. *SOURCETYPE* is interpreted as follows:

INPUT   The source bits come from *SOURCE*. *TEXTURE* is ignored.

INVERT   The source bits are the inverse of the bits from *SOURCE*. *TEXTURE* is ignored.

TEXTURE   The source bits come from *TEXTURE*. *SOURCE*, *SOURCELEFT*, and *SOURCEBOTTOM* are ignored.

OPERATION specifies how the source bits (as specified by *SOURCETYPE*) are combined with the bits in *DESTINATION* and stored back into *DESTINATION*. *DESTINATION* is one of the following:

REPLACE   All source bits (on or off) replace destination bits.

PAINT   Any source bits that are on replace the corresponding destination bits. Source bits that are off have no effect. Does a logical `OR` between the source bits and the destination bits.

INVERT   Any source bits that are on invert the corresponding destination bits. Does a logical `XOR` between the source bits and the destination bits.

ERASE   Any source bits that are on erase the corresponding destination bits. Does a logical `AND` operation between the inverse of the source bits and the destination bits.

Different combinations of *SOURCETYPE* and *OPERATION* can be specified to achieve many different effects. Given the following bitmaps as the values of *SOURCE*, *TEXTURE*, and *DESTINATION*:

SOURCE   TEXTURE   DESTINATION

`BITBLT` would produce the results given below for the difference combinations of *SOURCETYPE* and *OPERATION* (assuming *CLIPPINGREGION*, *SOURCELEFT*, etc. are set correctly, of course):

type= INPUT   type= INPUT   type= INPUT   type= INPUT
op= REPLACE   op= PAINT   op= INVERT   op= ERASE

type= INVERT op= REPLACE type= INVERT op= PAINT type= INVERT op= INVERT type= INVERT op= ERASE

type= TEXTURE op= REPLACE type= TEXTURE op= PAINT type= TEXTURE op= INVERT type= TEXTURE op= ERASE
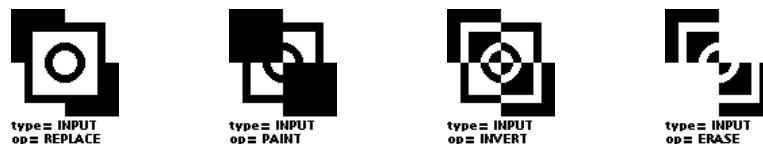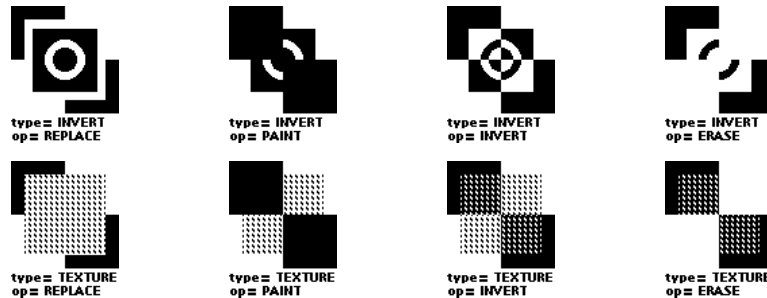
(**BLTSHADE** *TEXTURE DESTINATION DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT OPERATION CLIPPINGREGION*) [Function]

> BLTSHADE is the *SOURCETYPE = TEXTURE* case of BITBLT. It fills the specified region of the destination bitmap *DESTINATION* with the texture *TEXTURE*. *DESTINATION* can be a bitmap or image stream.

(**BITMAPIMAGESIZE** *BITMAP DIMENSION STREAM*) [Function]

> Returns the size that *BITMAP* will be when BITBLTed to *STREAM*, in *STREAM*'s units. *DIMENSION* can be one of *WIDTH*, *HEIGHT*, or NIL, in which case the dotted pair (*WIDTH . HEIGHT*) will be returned.

Drawing Lines

Interlisp-D provides several functions for drawing lines and curves on image streams. The line drawing functions are intended for interactive applications where efficiency is important. They do not allow the use of "brush" patterns, like the curve drawing functions, but (for display streams) they support drawing a line in INVERT mode, so redrawing the line will erase it. DRAWCURVE can be used to draw lines using a brush.

(**DRAWLINE** X_1 Y_1 X_2 Y_2 *WIDTH OPERATION STREAM COLOR DASHING*) [Function]

> Draws a straight line from the point (X_1, Y_1) to the point (X_2, Y_2) on the image stream *STREAM*. The position of *STREAM* is set to (X_2, Y_2). If X_1 equals X_2 and Y_1 equals Y_2, a point is drawn at (X_1, Y_1).

> *WIDTH* is the width of the line, in the units of the device. If *WIDTH* is NIL, the default is 1.

> *OPERATION* is the BITBLT operation used to draw the line. If *OPERATION* is NIL, the value of DSPOPERATION for the image stream is used.

> *COLOR* is a color specification that determines the color used to draw the line for image streams that support color. If *COLOR* is NIL, the *DSPCOLOR* of *STREAM* is used.

DASHING is a list of positive integers that determines the dashing characteristics of the line. The line is drawn for the number of points indicated by the first element of the dashing list, is not drawn for the number of points indicated by the second element. The third element indicates how long it will be on again, and so forth. The dashing sequence is repeated from the beginning when the list is exhausted. A brush LINEWITHBRUSH–by–LINEWITHBRUSH is used.

If *DASHING* is NIL, the line is not dashed.

(**DRAWBETWEEN** *POSITION$_1$ POSITION$_2$ WIDTH OPERATION STREAM COLOR DASHING*)
[Function]

Draws a line from the point *POSITION$_1$* to the point *POSITION$_2$* onto the destination bitmap of *STREAM*. The position of *STREAM* is set to *POSITION$_2$*.

In the Medley release, when using the color argument, Interpress **DRAWLINE** treats 16x16 bitmaps or negative numbers as shades/textures. Positive numbers continue to refer to color maps, and so cannot be used as textures. To convert an integer shade into a negative number use NEGSHADE (e.g. (NEGSHADE 42495) is –23041).

(**DRAWTO** *X Y WIDTH OPERATION STREAM COLOR DASHING*)   [Function]

Draws a line from the current position to the point (*X,Y*) onto the destination bitmap of *STREAM*. The position of *STREAM* is set to (*X,Y*).

(**RELDRAWTO** *DX DY WIDTH OPERATION STREAM COLOR DASHING*)   [Function]

Draws a line from the current position to the point (*DX,DY*) coordinates away onto the destination bitmap of *STREAM*. The position of *STREAM* is set to the end of the line. If *DX* and *DY* are both 0, nothing is drawn.

Drawing Curves

A curve is drawn by placing a brush pattern centered at each point along the curve's trajectory. A brush pattern is defined by its shape, size, and color. The predefined brush shapes are ROUND, SQUARE, HORIZONTAL, VERTICAL, and DIAGONAL; new brush shapes can be created using the INSTALLBRUSH function, described below. A brush size is an integer specifying the width of the brush in the units of the device. The color is a color specification, which is only used if the curve is drawn to an image stream that supports colors.

A brush is specified to the various drawing functions as a list of the form (SHAPE WIDTH COLOR), for example (SQUARE 2) or (VERTICAL 4 RED). A brush can also be specified as a positive integer, which is interpreted as a ROUND brush of that width. If a brush is a litatom, it is assumed to be a function which is called at each point of the curve's trajectory (with three arguments: the X-

coordinate of the point, the Y-coordinate, and the image stream), and should do whatever image stream operations are necessary to draw each point. Finally, if a brush is specified as NIL, a (ROUND 1) brush is used as default.

The appearance of a curve is also determined by its dashing characteristics. Dashing is specified by a list of positive integers. If a curve is dashed, the brush is placed along the trajectory for the number of units indicated by the first element of the dashing list. The brush is off, not placed in the bitmap, for a number of units indicated by the second element. The third element indicates how long it will be on again, and so forth. The dashing sequence is repeated from the beginning when the list is exhausted. The units used to measure dashing are the units of the brush. For example, specifying the dashing as (1 1) with a brush of (ROUND 16) would put the brush on the trajectory, skip 16 points, and put down another brush. A curve is not dashed if the dashing argument to the drawing function is NIL.

The curve functions use the image stream's clipping region and operation. Most types of image streams only support the PAINT operation when drawing curves. When drawing to a display stream, the curve-drawing functions accept the operation INVERT if the brush argument is 1. For brushes larger than 1, these functions will use the ERASE operation instead of INVERT. For display streams, the curve-drawing functions treat the REPLACE operation the same as PAINT.

(**DRAWCURVE** *KNOTS CLOSED BRUSH DASHING STREAM*)   [Function]

> Draws a "parametric cubic spline curve" on the image stream *STREAM*. *KNOTS* is a list of positions to which the curve will be fitted. If *CLOSED* is non-NIL, the curve will be closed; otherwise it ends at the first and last positions in *KNOTS*. *BRUSH* and *DASHING* are interpreted as described above.

> > For example,

> > ```
(DRAWCURVE '((10 . 10)(50 . 50)(100 . 10)(150 . 50))
   NIL '(ROUND 5) '(1 1 1 2) XX)
```

> would draw a curve like the following on the display stream XX:
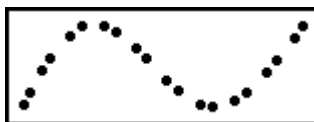


(**DRAWCIRCLE** *CENTERX CENTERY RADIUS BRUSH DASHING*
  *STREAM*)                                                        [Function]

> Draws a circle of radius *RADIUS* about the point (*CENTERX,CENTERY*) onto the image stream *STREAM*. *STREAM*'s position is left at (*CENTERX,CENTERY*). The other arguments are interpreted as described above.

(**DRAWARC** *CENTERX CENTERY RADIUS STARTANGLE NDEGREES BRUSH DASHINGSTREAM*) [Function]

> Draws an arc of the circle whose center point is (*CENTERX CENTERY*) and whose radius is *RADIUS* from the position at *STARTANGLE* degrees for *NDEGREES* number of degrees. If *STARTANGLE* is 0, the starting point will be (*CENTERX* (*CENTERY* + *RADIUS*)). If *NDEGREES* is positive, the arc will be counterclockwise. If *NDEGREES* is negative, the arc will be clockwise. The other arguments are interpreted as described in **DRAWCIRCLE**.

(**DRAWELLIPSE** *CENTERX CENTERY SEMIMINORRADIUS SEMIMAJORRADIUS ORIENTATION BRUSH DASHING*) *STREAM* [Function]

> Draws an ellipse with a minor radius of *SEMIMINORRADIUS* and a major radius of *SEMIMAJORRADIUS* about the point (*CENTERX,CENTERY*) onto the image stream *STREAM*. *ORIENTATION* is the angle of the major axis in degrees, positive in the counterclockwise direction. *STREAM*'s position is left at (*CENTERX,CENTERY*). The other arguments are interpreted as described above.

New brush shapes can be defined using the following function:

(**INSTALLBRUSH** *BRUSHNAME BRUSHFN BRUSHARRAY*) [Function]

> Installs a new brush called *BRUSHNAME* with creation-function *BRUSHFN* and optional array *BRUSHARRAY*. *BRUSHFN* should be a function of one argument (a width), which returns a bitmap of the brush for that width. *BRUSHFN* will be called to create new instances of *BRUSHNAME*-type brushes; the sixteen smallest instances will be pre-computed and cached. "Hand-crafted" brushes can be supplied as the *BRUSHARRAY* argument. Changing an existing brush can be done by calling INSTALLBRUSH with new *BRUSHFN* and/or *BRUSHARRAY*.

(**DRAWPOINT** *X Y BRUSH STREAM OPERATION*) [Function]

> Draws *BRUSH* centered around point (*X, Y*) on *STREAM*, using the operation *OPERATION*. *BRUSH* may be a bitmap or a brush.

## Miscellaneous Drawing and Printing Operations

(**DSPFILL** *REGION TEXTURE OPERATION STREAM*) [Function]

> Fills *REGION* of the image stream *STREAM* (within the clipping region) with the texture *TEXTURE*. If *REGION* is NIL, the whole clipping region of *STREAM* is used. If *TEXTURE* or *OPERATION* is NIL, the values for *STREAM* are used.

(**DRAWPOLYGON** *POINTS CLOSED BRUSH DASHING STREAM*)　　　　　[Function]

Draws a polygon on the image stream *STREAM*. *POINTS* is a list of positions to which the figure will be fitted (the vertices of the polygon). If *CLOSED* is non-NIL, then the starting position is specified only once in *POINTS*. If *CLOSED* is NIL, then the starting vertex must be specified twice in *POINTS*. *BRUSH* and *DASHING* are interpreted as described in Chapter 27 of the Interlisp-D Reference Manual.

For example,

```
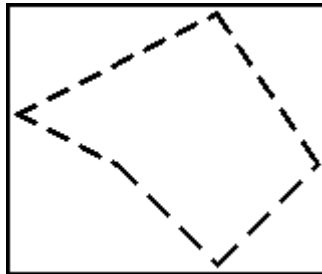(DRAWPOLYGON '((100 . 100) (50 . 125)
               (150 . 175) (200 . 100) (150 .
50))
               T '(ROUND 3) '(4 2) XX)
```

will draw a polygon like the following on the display stream XX.

(**FILLPOLYGON** *POINTS TEXTURE OPERATION WINDNUMBER STREAM*)　　　[Function]

OPERATION is the **BITBLT** operation (see page 27.15 in the Interlisp-D Reference Manual) used to fill the polygon. If the *OPERATION* is NIL, the *OPERATION* defaults to the STREAM default *OPERATION*.

WINDNUMBER is the number for the winding rule convention . This number is either 0 or 1; 0 indicates the "zero" winding rule, 1 indicates the "odd" winding rule.

When filling a polygon, there is more than one way of dealing with the situation where two polygon sides intersect, or one polygon is fully inside the other. Currently, **FILLPOLYGON** to a display stream uses the "odd" winding rule, which means that intersecting polygon sides define areas that are filled or not filled somewhat like a checkerboard. For example,

```
(FILLPOLYGON
  '( ((110 . 110)(150 . 200)(190 . 110))
     ((135 . 125)(160 . 125)(160 . 150)(135 .
150)) )
    GRAYSHADE WINDOW)
```

will produce a display something like this:

This fill convention also takes into account all polygons in *POINTS*, if it specifies multiple polygons.

(**FILLCIRCLE** *CENTERX CENTERY RADIUS TEXTURE STREAM*)   [Function]

Fills in a circular area of radius *RADIUS* about the point (*CENTERX*,*CENTERY*) in *STREAM* with *TEXTURE*. *STREAM*'s position is left at (*CENTERX*,*CENTERY*).

(**DSPRESET** *STREAM*)   [Function]

Sets the X coordinate of *STREAM* to its left margin, sets its Y coordinate to the top of the clipping region minus the font ascent. For a display stream, this also fills its destination bitmap with its background texture.

(**DSPNEWPAGE** *STREAM*)   [Function]

Starts a new page. The X coordinate is set to the left margin, and the Y coordinate is set to the top margin plus the linefeed.

(**CENTERPRINTINREGION** *EXP REGION STREAM*)   [Function]

Prints *EXP* so that is it centered within *REGION* of the *STREAM*. If *REGION* is NIL, *EXP* will be centered in the clipping region of *STREAM*.

Drawing and Shading Grids

A grid is a partitioning of an arbitrary coordinate system (hereafter referred to as the "source system") into rectangles. This section describes functions that operate on grids. It includes functions to draw the outline of a grid, to translate between positions in a source system and grid coordinates (the coordinates of the rectangle which contains a given position), and to shade grid rectangles. A grid is defined by its "unit grid," a region (called a grid specification) which is the origin rectangle of the grid in terms of the source system. Its LEFT field is interpreted as the X-coordinate of the left edge of the origin rectangle, its BOTTOM field is the Y-coordinate of the bottom edge of the origin rectangle, its WIDTH is the width of the grid rectangles, and its HEIGHT is the height of the grid rectangles.

(**GRID** *GRIDSPEC WIDTH HEIGHT BORDER STREAM GRIDSHADE*)   [Function]

Outlines the grid defined by *GRIDSPEC* which is *WIDTH* rectangles wide and *HEIGHT* rectangles high on *STREAM*. Each box in the grid has a border within it that is *BORDER* points on each side; so the resulting lines in the grid are 2*BORDER thick. If *BORDER* is the atom *POINT*, instead of a border the lower left point of each grid rectangle will be turned

on. If *GRIDSHADE* is non-NIL, it should be a texture and the border lines will be drawn using that texture.

(**SHADEGRIDBOX** *X Y SHADE OPERATION GRIDSPEC GRIDBORDER STREAM*)   [Function]

Shades the grid rectangle (*X,Y*) of *GRIDSPEC* with texture *SHADE* using *OPERATION* on *STREAM*. *GRIDBORDER* is interpreted the same as for GRID.

The following two functions map from the X,Y coordinates of the source system into the grid X,Y coordinates:

(**GRIDXCOORD** *XCOORD GRIDSPEC*)   [Function]

Returns the grid X-coordinate (in the grid specified by *GRIDSPEC*) that contains the source system X-coordinate *XCOORD*.

(**GRIDYCOORD** *YCOORD GRIDSPEC*)   [Function]

Returns the grid Y-coordinate (in the grid specified by *GRIDSPEC*) that contains the source system Y-coordinate *YCOORD*.

The following two functions map from the grid X,Y coordinates into the X,Y coordinates of the source system:

(**LEFTOFGRIDCOORD** *GRIDX GRIDSPEC*)   [Function]

Returns the source system X-coordinate of the left edge of a grid rectangle at grid X-coordinate *GRIDX* (in the grid specified by *GRIDSPEC*).

(**BOTTOMOFGRIDCOORD** *GRIDY GRIDSPEC*)   [Function]

Returns the source system Y-coordinate of the bottom edge of a grid rectangle at grid Y-coordinate *GRIDY* (in the grid specified by *GRIDSPEC*).

Display Streams

Display streams (image streams of type DISPLAY) are used to control graphic output operations to a bitmap, known as the "destination" bitmap of the display stream. For each window on the screen, there is an associated display stream which controls graphics operations to a specific part of the screen bitmap. Any of the functions that take a display stream will also take a window, and use the associated display stream. Display streams can also have a destination bitmap that is not connected to any window or display device.

(**DSPCREATE** *DESTINATION*) [Function]

Creates and returns a display stream. If *DESTINATION* is specified, it is used as the destination bitmap, otherwise the screen bitmap is used.

(**DSPDESTINATION** *DESTINATION DISPLAYSTREAM*) [Function]

Returns the current destination bitmap for *DISPLAYSTREAM*, setting it to *DESTINATION* if non-NIL. *DESTINATION* can be either the screen bitmap, or an auxilliary bitmap in order to construct figures, possibly save them, and then display them in a single operation.

> Warning: The window system maintains the destination of a window's display stream. Users should be very careful about changing this field.

(**DSPXOFFSET** *XOFFSET DISPLAYSTREAM*) [Function]

(**DSPYOFFSET** *YOFFSET DISPLAYSTREAM*) [Function]

Each display stream has its own coordinate system, separate from the coordinate system of its destination bitmap. Having the coordinate system local to the display stream allows objects to be displayed at different places by translating the display stream's coordinate system relative to its destination bitmap. This local coordinate system is defined by the X offset and Y offset.

DSPXOFFSET returns the current X offset for *DISPLAYSTREAM*, the X origin of the display stream's coordinate system in the destination bitmap's coordinate system. It is set to *XOFFSET* if non-NIL.

DSPYOFFSET returns the current Y offset for *DISPLAYSTREAM*, the Y origin of the display stream's coordinate system in the destination bitmap's coordinate system. It is set to *YOFFSET* if non-NIL.

The X offset and Y offset for a display stream are both initially 0 (no X or Y-coordinate translation).

Warning: The window system maintains the X and Y offset of a window's display stream. Users should be very careful about changing these fields.

(**DSPTEXTURE** *TEXTURE DISPLAYSTREAM*) [Function]

Returns the current texture used as the background pattern for *DISPLAYSTREAM*. It is set to *TEXTURE* if non-NIL. Initially the value of WHITESHADE.

(**DSPSOURCETYPE** *SOURCETYPE DISPLAYSTREAM*) [Function]

Returns the current BITBLT sourcetype used when printing characters to the display stream. It is set to *SOURCETYPE*, if non-NIL. Must be either INPUT or INVERT. Initially INPUT.

(**DSPSCROLL** *SWITCHSETTING DISPLAYSTREAM*)   [Function]

> Returns the current value of the "scroll flag," a flag that determines the scrolling behavior of the display stream; either ON or OFF. If ON, the bits in the display streams's destination bitmap are moved after any linefeed that moves the current position out of the destination bitmap. Any bits moved out of the current clipping region are lost. Does not adjust the X offset, Y offset, or clipping region of the display stream. Initially OFF.

> Sets the scroll flag to *SWITCHSETTING*, if non-NIL.

> Note: The word "scrolling" also describes the use of "scroll bars" on the left and bottom of a window to move an object displayed in a window.

Each window has an associated display stream. To get the window of a particular display stream, use WFROMDS:

(**WFROMDS** *DISPLAYSTREAM DONTCREATE*)   [Function]

> Returns the window associated with *DISPLAYSTREAM*, creating a window if one does not exist (and *DONTCREATE* is NIL). Returns NIL if the destination of *DISPLAYSTREAM* is not a screen bitmap that supports a window system.

> If *DONTCREATE* is non-NIL, WFROMDS will never create a window, and returns NIL if *DISPLAYSTREAM* does not have an associated window.

> TTYDISPLAYSTREAM calls WFROMDS with DONTCREATE = T, so it will not create a window unnecessarily. Also, if WFROMDS does create a window, it calls CREATEW with NOOPENFLG = T.

(**DSPBACKUP** *WIDTH DISPLAYSTREAM*)   [Function]

> Backs up *DISPLAYSTREAM* over a character which is *WIDTH* screen points wide. DSPBACKUP fills the backed over area with the display stream's background texture and decreases the X position by *WIDTH*. If this would put the X position less than *DISPLAYSTREAM*'s left margin, its operation is stopped at the left margin. It returns T if any bits were written, NIL otherwise.

Fonts

A font is the collection of images that are printed or displayed when characters are output to a graphic output device. Some simple displays and printers can only print characters using one font. Bitmap displays and graphic printers can print characters using a large number of fonts.

Fonts are identified by a distinctive style or family (such as Modern or Classic), a size (such as 10 points), and a face (such as bold or italic). Fonts also have a rotation that indicates the orientation of characters on the screen or page. A normal horizontal font (also called a portrait font) has a rotation of 0; the rotation of a vertical (landscape) font is 90 degrees. While any combination can be specified, in

practice the user will find that only certain combinations of families, sizes, faces, and rotations are available for any graphic output device.

To specify a font to the functions described below, a `FAMILY` is represented by a literal atom, a `SIZE` by a positive integer, and a `FACE` by a three-element list of the form (`WEIGHT   SLOPE EXPANSION`). `WEIGHT`, which indicates the thickness of the characters, can be `BOLD`, `MEDIUM`, or `LIGHT`; `SLOPE` can be `ITALIC` or `REGULAR`; and `EXPANSION` can be `REGULAR`, `COMPRESSED`, or `EXPANDED`, indicating how spread out the characters are. For convenience, faces may also be specified by three-character atoms, where each character is the first letter of the corresponding field. Thus, `MRR` is a synonym for (`MEDIUM  REGULAR  REGULAR`). In addition, certain common face combinations may be indicated by special literal atoms:

```
STANDARD = (MEDIUM REGULAR REGULAR) = MRR
ITALIC = (MEDIUM ITALIC REGULAR) = MIR
BOLD = (BOLD REGULAR REGULAR) = BRR
BOLDITALIC = (BOLD ITALIC REGULAR) = BIR
```

Interlisp represents all the information related to a font in an object called a font descriptor. Font descriptors contain the family, size, etc. properties used to represent the font. In addition, for each character in the font, the font descriptor contains width information for the character and (for display fonts) a bitmap containing the picture of the character.

The font functions can take fonts specified in a variety of different ways. `DSPFONT`, `FONTCREATE`, `FONTCOPY`, etc. can be applied to font descriptors, "font lists" such as '(`MODERN 10`), image streams (coerced to its current font), or windows (coerced to the current font of its display stream). The printout command ".`FONT`" will also accept fonts specified in any of these forms.

In general font files use the following format:

The family name (e.g., Modern); a two digit size (e.g., 08); a three letter Face (e.g., BIR, for Bold Italic Regular); the letter C followed by the font's character set in base 8 (e.g., C41); and finally an extension (e.g., Displayfont).
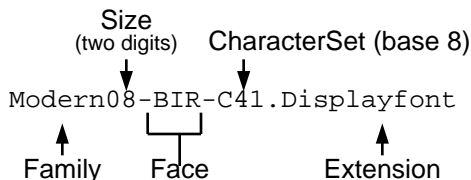
(**FONTCREATE** *FAMILY SIZE FACE ROTATION DEVICE NOERRORFLG CHARSET*)
  [Function]

> Returns a font descriptor for the specified font. *FAMILY* is a litatom specifying the font family. *SIZE* is an integer indicating the size of the font in points. *FACE* specifies the face characteristics in one of the formats listed above; if *FACE* is NIL, STANDARD is used. *ROTATION*, which specifies the orientation of the font, is 0 (or NIL) for a portrait font and 90 for a landscape font. *DEVICE* indicates the output device for the font, and can be any image stream type , such as DISPLAY, INTERPRESS, etc. *DEVICE* may also be an image stream, in which case the type of the stream determines the font device. *DEVICE* defaults to DISPLAY.

> The *FAMILY* argument to FONTCREATE may also be a list, in which case it is interpreted as a font-specification quintuple, a list of the form (*FAMILY SIZE FACE ROTATION DEVICE*). Thus, (FONTCREATE '(GACHA 10 BOLD)) is equivalent to (FONTCREATE 'GACHA 10 'BOLD). *FAMILY* may also be a font descriptor, in which case that descriptor is simply returned.

> If a font descriptor has already been created for the specified font, FONTCREATE simply returns it. If it has not been created, FONTCREATE has to read the font information from a font file that contains the information for that font. The name of an appropriate font file, and the algorithm for searching depends on the device that the font is for, and is described in more detail below. If an appropriate font file is found, it is read into a font descriptor. If no file is found, for DISPLAY fonts FONTCREATE looks for fonts with less face information and fakes the remaining faces (such as by doubling the bit pattern of each character or slanting it). For hardcopy printer fonts, there is no acceptable faking algorithm.

> If no acceptable font is found, the action of FONTCREATE is determined by NOERRORFLG. If NOERRORFLG is NIL, it generates a FONT NOT FOUND error with the offending font specification; otherwise, FONTCREATE returns NIL.

> CHARSET is the character set which will be read to create the font. Defaults to 0. For more information on character sets, see NS Characters.

(**FONTP** *X*)   [Function]

> Returns *X* if *X* is a font descriptor; NIL otherwise.

(**FONTPROP** *FONT PROP*)   [Function]

> Returns the value of the *PROP* property of font *FONT*. The following font properties are recognized:

> FAMILY   The style of the font, represented as a literal atom, such as CLASSIC or MODERN.

> SIZE   A positive integer giving the size of the font, in printer's points (1/72 of an inch).

WEIGHT	The thickness of the characters; one of BOLD, MEDIUM, or LIGHT.
SLOPE	The "slope" of the characters in the font; one of ITALIC or REGULAR.
EXPANSION	The extent to which the characters in the font are spread out; one of REGULAR, COMPRESSED, or EXPANDED. Most available fonts have EXPANSION = REGULAR.
FACE	A three-element list of the form (WEIGHT SLOPE EXPANSION), giving all of the typeface parameters.
ROTATION	An integer that gives the orientation of the font characters on the screen or page, in degrees. A normal horizontal font (also called a portrait font) has a rotation of 0; the rotation of a vertical (landscape) font is 90.
DEVICE	The device that the font can be printed on; one of DISPLAY, INTERPRESS, etc.
ASCENT	An integer giving the maximum height of any character in the font from its base line (the printing position). The top line will be at BASELINE+ASCENT−1.
DESCENT	An integer giving the maximum extent of any character below the base line, such as the lower part of a "p". The bottom line of a character will be at BASELINE−DESCENT.
HEIGHT	Equal to ASCENT + DESCENT.
SPEC	The (FAMILY SIZE FACE ROTATION DEVICE) quintuple by which the font is known to Lisp.
DEVICESPEC	The (FAMILY SIZE FACE ROTATION DEVICE) quintuple that identifies what will be used to represent the font on the display or printer. It will differ from the SPEC property only if an implicit coercion is done to approximate the specified font with one that actually exists on the device.
SCALE	The units per printer's point (1/72 of an inch) in which the font is measured. For example, this is 35.27778 (the number of micas per printer's point) for Interpress fonts, which are measured in terms of micas.

(**FONTCOPY** *OLDFONT* *PROP$_1$* *VAL$_1$* *PROP$_2$* *VAL$_2$* ...)   [NoSpread Function]

Returns a font descriptor that is a copy of the font *OLDFONT*, but which differs from *OLDFONT* in that *OLDFONT*'s properties are replaced by the specified properties and values. Thus, (FONTCOPY FONT 'WEIGHT 'BOLD 'DEVICE 'INTERPRESS) will return a bold Interpress font with all other properties the same as those of FONT. FONTCOPY accepts the properties FAMILY, SIZE, WEIGHT, SLOPE, EXPANSION, FACE, ROTATION, and DEVICE. If the first property is a list, it is taken to be the *PROP$_1$* *VAL$_1$* *PROP$_2$* *VAL$_2$* ... sequence. Thus, (FONTCOPY FONT '(WEIGHT BOLD DEVICE INTERPRESS)) is equivalent to the example above.

If the property NOERROR is specified with value non-NIL, FONTCOPY will return NIL rather than causing an error if the specified font cannot be created.

(**FONTSAVAILABLE** *FAMILY SIZE FACE ROTATION DEVICE CHECKFILESTOO?*) [Function]

Returns a list of available fonts that match the given specification. *FAMILY, SIZE, FACE, ROTATION*, and *DEVICE* are the same as for FONTCREATE. Additionally, any of them can be the atom *, in which case all values of that field are matched.

If *CHECKFILESTOO?* is NIL, only fonts already loaded into virtual memory will be considered. If *CHECKFILESTOO?* is non-NIL, the font directories for the specified device will be searched. When checking font files, the *ROTATION* is ignored.

> Note: The search is conditional on the status of the server which holds the font. Thus a file server crash may prevent FONTCREATE from finding a file that an earlier FONTSAVAILABLE returned.

Each element of the list returned will be of the form (*FAMILY SIZE FACE ROTATION DEVICE*).

Examples:

(FONTSAVAILABLE 'MODERN 10 'MRR 0 'DISPLAY)

will return ((MODERN 10 (MEDIUM REGULAR REGULAR) 0 DISPLAY)) if the regular Modern 10 font for the display is in virtual memory; NIL otherwise.

(FONTSAVAILABLE '* 14 '* '* 'INTERPRESS T)

will return a list of all the size 14 Interpress fonts, whether they are in virtual memory or in font files.

(**SETFONTDESCRIPTOR** *FAMILY SIZE FACE ROTATION DEVICE FONT*) [Function]

Indicates to the system that *FONT* is the font that should be associated with the *FAMILY SIZE FACE ROTATION DEVICE* characteristics. If *FONT* is NIL, the font associated with these characteristics is cleared and will be recreated the next time it is needed. As with FONTPROP and FONTCOPY, *FONT* is coerced to a font descriptor if it is not one already.

This functions is useful when it is desirable to simulate an unavailable font or to use a font with characteristics different from the interpretations provided by the system.

(**DEFAULTFONT** *DEVICE FONT* —) [Function]

Returns the font that would be used as the default (if NIL were specified as a font argument) for image stream type *DEVICE*. If *FONT* is a font descriptor, it is set to be the default font for *DEVICE*.

(**CHARWIDTH** *CHARCODE FONT*) [Function]

> *CHARCODE* is an integer that represents a valid character (as returned by CHCON1). Returns the amount by which an image stream's **X**-position will be incremented when the character is printed.

(**CHARWIDTHY** *CHARCODE FONT*) [Function]

> Like CHARWIDTH, but returns the **Y** component of the character's width, the amount by which an image stream's **Y**-position will be incremented when the character is printed. This will be zero for most characters in normal portrait fonts, but may be non-zero for landscape fonts or for vector-drawing fonts.

(**STRINGWIDTH** *STR FONT FLG RDTBL*) [Function]

> Returns the amount by which a stream's **X**-position will be incremented if the printname for the Interlisp-D object *STR* is printed in font *FONT*. If *FONT* is NIL, DEFAULTFONT is used as *FONT*. lf *FONT* is an image stream, its font is used. If *FLG* is non-NIL, the PRIN2-pname of *STR* with respect to the readtable *RDTBL* is used.

(**STRINGREGION** *STR STREAM PRIN2FLG RDTBL*) [Function]

> Returns the region occupied by *STR* if it were printed at the current location in the image stream *STREAM*. This is useful, for example, for determining where text is in a window to allow the user to select it. The arguments *PRIN2FLG* and *RDTBL* are passed to STRINGWIDTH.

> > Note: STRINGREGION does not take into account any carriage returns in the string, or carriage returns that may be automatically printed if *STR* is printed to *STREAM*. Therefore, the value returned is meaningless for multi-line strings.

The following functions allow the user to access and change the bitmaps for individual characters in a display font. Note: Character code 256 can be used to access the "dummy" character, used for characters in the font with no bitmap defined.

(**GETCHARBITMAP** *CHARCODE FONT*) [Function]

> Returns a bitmap containing a copy of the image of the character *CHARCODE* in the font *FONT*.

(**PUTCHARBITMAP** *CHARCODE FONT NEWCHARBITMAP NEWCHARDESCENT*) [Function]

> Changes the bitmap image of the character *CHARCODE* in the font *FONT* to the bitmap *NEWCHARBITMAP*. If *NEWCHARDESCENT* is non-NIL, the descent of the character is changed to the value of *NEWCHARDESCENT*.

(**EDITCHAR** *CHARCODE FONT*) [Function]

Calls the bitmap editor (EDITBM) on the bitmap image of the character *CHARCODE* in the font *FONT*. *CHARCODE* can be a character code (as returned by CHCON1) or an atom or string, in which case the first character of *CHARCODE* is used.

(**WRITESTRIKEFONTFILE** *FONT CHARSET FILENAME*) [Function]

Takes a display font font descriptor and a character set number, and writes that character set into a file suitable for reading in again. Note that the font descriptor's current state is used (which was perhaps modified by INSPECTing the datum), so this provides a mechanism for creating/modifying new fonts.

For example:

```
(WRITESTRIKEFONTFILE (FONTCREATE 'GACHA 10) 0 '{DSK}Magic10-
MRR-C0.DISPLAYFONT)
```

If your DISPLAYFONTDIRECTORIES includes {DSK}, then a subsequent (FONTCREATE 'MAGIC 10) will create a new font descriptor whose appearance is the same as the old Gacha font descriptor.

However, the new font is identical to the old one in appearance only. The individual datatype fields and bitmap may not be the same as those in the old font descriptor, due to peculiarities of different font file formats.

Font Files and Font Directories

If FONTCREATE is called to create a font that has not been loaded into Interlisp, FONTCREATE has to read the font information from a font file that contains the information for that font. For printer devices, the font files have to contain width information for each character in the font. For display fonts, the font files have to contain, in addition, bitmap images for each character in the fonts. The font file names, formats, and searching algorithms are different for each device. There are a set of variables for each device, that determine the directories that are searched for font files. All of these variables must be set before Interlisp can auto-load font files. These variables should be initialized in the site-specific INIT file.

DISPLAYFONTDIRECTORIES [Variable]

Value is a list of directories searched to find font bitmap files for display fonts.

DISPLAYFONTEXTENSIONS [Variable]

Value is a list of file extensions used when searching DISPLAYFONTDIRECTORIES for display fonts. Initially set to (DISPLAYFONT), but when using older font files it may be necessary to add STRIKE and AC to this list.

INTERPRESSFONTDIRECTORIES [Variable]

Value is a list of directories searched to find font widths files for Interpress fonts.

PRESSFONTWIDTHSFILES [Variable]

Value is a list of files (not directories) searched to find font widths files for Press fonts. Press font widths are packed into large files (usually named `FONTS.WIDTHS`).

Font Profiles

`PRETTYPRINT` contains a facility for printing different elements (user functions, system functions, clisp words, comments, etc.) in different fonts to emphasize (or deemphasize) their importance, and in general to provide for a more pleasing appearance. Of course, in order to be useful, this facility requires that the user is printing on a device (such as a bitmapped display or a laser printer) which supports multiple fonts.

`PRETTYPRINT` signals font changes by inserting into the file a user-defined escape sequence (the value of the variable `FONTESCAPECHAR`) followed by the character code which specifies, by number, which font to use, i.e. ↑A for font number 1, etc. Thus, if `FONTESCAPECHAR` were the character ↑F, ↑F↑C would be output to change to font 3, ↑F↑A to change to font 1, etc. If `FONTESCAPECHAR` consists of characters which are separator charactors in `FILERDTBL`, then a file with font changes in it can also be loaded back in.

Currently, `PRETTYPRINT` uses the following font classes. The user can specify separate fonts for each of these classes, or use the same font for several different classes.

LAMBDAFONT	The font for printing the name of the function being prettyprinted, before the actual definition (usually a large font).
CLISPFONT	If `CLISPFLG` is on, the font for printing any clisp words, i.e. atoms with property `CLISPWORD`.
COMMENTFONT	The font used for comments.
USERFONT	The font for the name of any function in the file, or any member of the list `FONTFNS`.
SYSTEMFONT	The font for any other (defined) function.
CHANGEFONT	The font for an expression marked by the editor as having been changed.
PRETTYCOMFONT	The font for the operand of a file package command.
DEFAULTFONT	The font for everything else.

Note that not all combinations of fonts will be aesthetically pleasing (or even readable!) and the user may have to experiment to find a compatible set.

Although in some implementations LAMBDAFONT et al. may be defined as variables, one should not set them directly, but should indicate what font is to be used for each class by calling the function FONTPROFILE:

(**FONTPROFILE** *PROFILE*)   [Function]

> Sets up the font classes as determined by *PROFILE*, a list of elements which defines the correspondence between font classes and specific fonts. Each element of *PROFILE* is a list of the form:
>
> > (FONTCLASS   FONT#   DISPLAYFONT   PRESSFONT
> > INTERPRESSFONT)
>
> FONTCLASS is the font class name and FONT# is the font number for that class. For each font class name, the escape sequence will consist of FONTESCAPECHAR followed by the character code for the font number, e.g. ↑A for font number 1, etc.
>
> If FONT# is NIL for any font class, the font class named DEFAULTFONT (which must always be specified) is used. Alternatively, if FONT# is the name of a previously defined font class, this font class will be equivalenced to the previously defined one.
>
> DISPLAYFONT, PRESSFONT, and INTERPRESSFONT are font specifications (of the form accepted by FONTCREATE) for the fonts to use when printing to the display and to Press and Interpress printers respectively.

FONTPROFILE   [Variable]

> This is the variable used to store the current font profile, in the form accepted by the function FONTPROFILE. Note that simply editing this value will not change the fonts used for the various font classes; it is necessary to execute (FONTPROFILE FONTPROFILE) to install the value of this variable.

The process of printing with multiple fonts is affected by a large number of variables: FONTPROFILE, FILELINELENGTH, PRETTYLCOM, etc. To facilitate switching back and forth between various sets of values for the font variables, Interlisp supports the idea of named "font configurations" encapsulating the values of all relevant variables.

To create a new font configuration, set all "relevant" variables to the values you want, and then call FONTNAME to save them (on the variable FONTDEFS) under a given name. To install a particular font configuration, call FONTSET giving it your name. To change the values in a saved font configuration, edit the value of the variable FONTDEFS.

Note: The list of variables saved by FONTNAME is stored in the variable FONTDEFSVARS. This can be changed by the user.

(**FONTSET** *NAME*) [Function]

> Installs font configuration for *NAME*. Also evaluates (FONTPROFILE FONTPROFILE) to install the font classes as specified in the new value of the variable FONTPROFILE. Generates an error if *NAME* not previously defined.

FONTDEFSVARS [Variable]

> The list of variables to be packaged by a FONTNAME. Initially FONTCHANGEFLG, FILELINELENGTH, COMMENTLINELENGTH, FIRSTCOL, PRETTYLCOM, LISTFILESTR, and FONTPROFILE.

FONTDEFS [Variable]

> An association list of font configurations. FONTDEFS is a list of elements of form (NAME . PARAMETER-PAIRS). To save a configuration on a file after performing a FONTNAME to define it, the user could either save the entire value of FONTDEFS, or use the ALISTS file package command to dump out just the one configuration.

FONTESCAPECHAR [Variable]

> The character or string used to signal the start of a font escape sequence.

FONTCHANGEFLG [Variable]

> If T, enables fonts when prettyprinting. If NIL, disables fonts. ALL indicates that all calls to CHANGEFONT are executed.

LISTFILESTR [Variable]

> In Interlisp-10, passed to the operating system by LISTFILES. Can be used to specify subcommands to the LIST command, e.g. to establish correspondance between font number and font name.

COMMENTLINELENGTH [Variable]

> Since comments are usually printed in a smaller font, COMMENTLINELENGTH is provided to offset the fact that Interlisp does not know about font widths. When FONTCHANGEFLG = T, CAR of COMMENTLINELENGTH is the linelength used to print short comments, i.e. those printed in the right margin, and CDR is the linelength used when printing full width comments.

(**CHANGEFONT** *FONT STREAM*) [Function]

> Executes the operations on *STREAM* to change to the font *FONT*. For use in PRETTYPRINTMACROS.

Image Objects

An Image Object is an object that includes information about an image, such as how to display it, how to print it, and how to manipulate it when it is included in a collection of images (such as a document). More generally, it enables you to include one kind of image, with its own semantics, layout rules, and editing paradigms, inside another kind of image. Image Objects provide a general-purpose interface between image users who want to manipulate arbitrary images, and image producers, who create images for use, say, in documents.

Images are encapsulated inside a uniform barrier—the `IMAGEOBJ` data type. From the outside, you communicate to the image by calling a standard set of functions. For example, calling one function tells you how big the image is; calling another causes the image object to be displayed where you tell it, and so on. Anyone who wants to create images for general use can implement his own brand of `IMAGEOBJ`. `IMAGEOBJ`s have been implemented (in library packages) for bitmaps, menus, annotations, graphs, and sketches.

Image Objects were originally implemented to support inserting images into TEdit text files, but the facility is available for use by any tools that manipulate images. The Image Object interface allows objects to exist in TEdit documents and be edited with their own editor. It also provides a facility in which objects can be shift-selected (or "copy-selected") between TEdit and non-TEdit windows. For example, the Image Objects interface allows you to copy-select graphs from a Grapher window into a TEdit window. The source window (where the object comes from) does not have to know what sort of window the destination window (where the object is inserted) is, and the destination does not have to know where the insertion comes from.

A new data type, `IMAGEOBJ`, contains the data and the procedures necessary to manipulate an object that is to be manipulated in this way. `IMAGEOBJ`s are created with the function `IMAGEOBJCREATE` (below).

Another new data type, `IMAGEFNS`, is a vector of the procedures necessary to define the behavior of a type of `IMAGEOBJ`. Grouping the operations in a separate data type allows multiple instances of the same type of image object to share procedure vectors. The data and procedure fields of an `IMAGEOBJ` have a uniform interface through the function `IMAGEOBJPROP`. `IMAGEFNS` are created with the function `IMAGEFNSCREATE`:

> (**IMAGEFNSCREATE** *DISPLAYFN IMAGEBOXFN PUTFN GETFN COPYFN BUTTONEVENTINFN COPYBUTTONEVENTINFN WHENMOVEDFN WHENINSERTEDFN WHENDELETEDFN WHENCOPIEDFN WHENOPERATEDONFN PREPRINTFN* — [Function]
>
> Returns an `IMAGEFNS` object that contains the functions necessary to define the behavior of an `IMAGEOBJ`.
>
> The arguments *DISPLAYFN* through *PREPRINTFN* should all be function names to be stored as the "methods" of the `IMAGEFNS`. The purpose of each `IMAGEFNS` method is described below.

Note: Image objects must be "registered" before they can be read by TEdit or HREAD. IMAGEFNSCREATE implicitly registers its GETFN argument.

(**IMAGEOBJCREATE** *OBJECTDATUM IMAGEFNS***)** [Function]

Returns an IMAGEOBJ that contains the object datum *OBJECTDATUM* and the operations vector *IMAGEFNS*. *OBJECTDATUM* can be arbitrary data.

(**IMAGEOBJPROP** *IMAGEOBJECT PROPERTY NEWVALUE*) [NoSpread Function]

Accesses and sets the properties of an IMAGEOBJ. Returns the current value of the *PROPERTY* property of the image object *IMAGEOBJECT*. If *NEWVALUE* is given, the property is set to it.

IMAGEOBJPROP can be used on the system properties OBJECTDATUM, DISPLAYFN, IMAGEBOXFN, PUTFN, GETFN, COPYFN, **BUTTONEVENTINFN**, COPYBUTTONEVENTINFN, WHENOPERATEDONFN, and PREPRINTFN. Additionally, it can be used to save arbitrary properties on an IMAGEOBJ.

(**IMAGEFNSP** *X*) [Function]

Returns *X* if *X* is an IMAGEFNS object, NIL otherwise.

(**IMAGEOBJP** *X*) [Function]

Returns *X* if *X* is an IMAGEOBJ object, NIL otherwise.

IMAGEFNS Methods

Note: Many of the IMAGEFNS methods below are passed "host stream" arguments. The TEdit text editor passes the "text stream" (an object contain all of the information in the document being edited) as the "host stream" argument. Other editing programs that want to use image objects may want to pass the data structure being edited to the IMAGEFNS methods as the "host stream" argument.

(**DISPLAYFN** *IMAGEOBJ IMAGESTREAM IMAGESTREAMTYPE HOSTSTREAM*) [IMAGEFNS Method]

The DISPLAYFN method is called to display the object *IMAGEOBJ* at the current position on *IMAGESTREAM*. The type of *IMAGESTREAM* indicates whether the device is the display or some other image stream.

Note: When the DISPLAYFN method is called, the offset and clipping regions for the stream are set so the object's image is at (0,0), and only that image area can be modified.

(**IMAGEBOXFN** *IMAGEOBJ IMAGESTREAM CURRENTX RIGHTMARGIN*) [IMAGEFNS Method]

The IMAGEBOXFN method should return the size of the object as an IMAGEBOX, which is a data structure that describes the image laid down when an *IMAGEOBJ* is displayed in terms of width, height, and descender height. An IMAGEBOX has four fields: XSIZE,

YSIZE, YDESC, and XKERN. XSIZE and YSIZE are the width and height of the object image. YDESC and XKERN give the position of the baseline and the left edge of the image relative to where you want to position it. For characters, the YDESC is the descent (height of the descender) and the XKERN is the amount of left kerning (note: TEdit doesn't support left kerning).

The IMAGEBOXFN looks at the type of the stream to determine the output device if the object's size changes from device to device. (For example, a bit-map object may specify a scale factor that is ignored when the bit map is displayed on the screen.) *CURRENTX* and *RIGHTMARGIN* allow an object to take account of its environment when deciding how big it is. If these fields are not available, they are NIL.

> Note: TEdit calls the IMAGEBOXFN only during line formatting, then caches the IMAGEBOX as the BOUNDBOX property of the *IMAGEOBJ*. This avoids the need to call the IMAGEBOXFN when incomplete position and margin information is available.

(**PUTFN** *IMAGEOBJ FILESTREAM*)   [IMAGEFNS Method]

The PUTFN method is called to save the object on a file. It prints a description on *FILESTREAM* that, when read by the corresponding GETFN method (see below), regenerates the image object. (TEdit and HPRINT take care of writing out the name of the GETFN.)

(**GETFN** *FILESTREAM*)   [IMAGEFNS Method]

The GETFN method is called when the object is encountered on the file during input. It reads the description that was written by the PUTFN method and returns an IMAGEOBJ.

(**COPYFN** *IMAGEOBJ SOURCEHOSTSTREAM TARGETHOSTSTREAM*)   [IMAGEFNS Method]

The COPYFN method is called during a copy-select operation. It should return a copy of *IMAGEOBJ*. If it returns the litatom DON'T, copying is suppressed.

(**BUTTONEVENTINFN** *IMAGEOBJ WINDOWSTREAM SELECTION RELX RELY WINDOW HOSTSTREAM BUTTON*)   [IMAGEFNS Method]

The BUTTONEVENTINFN method is called when you press a mouse button inside the object. The BUTTONEVENTINFN decides whether or not to handle the button, to track the cursor in parallel with mouse movement, and to invoke selections or edits supported by the object (but see the COPYBUTTONEVENTINFN method below). If the BUTTONEVENTINFN returns NIL, TEdit treats the button press as a selection at its level. Note that when this function is first called, a button is down. The BUTTONEVENTINFN should also support the button-down protocol to descend inside of any composite objects with in it. In most cases, the BUTTONEVENTINFN relinquishes control (i.e., returns) when the cursor leaves its object's region.

When the `BUTTONEVENTINFN` is called, the window's clipping region and offsets have been changed so that the lower-left corner of the object's image is at (0 , 0), and only the object's image can be changed. The selection is available for changing to fit your needs; the mouse button went down at (`RELX`,`RELY`) within the object's image. You can affect how TEdit treats the selection by returning one of several values. If you return `NIL`, TEdit forgets that you selected an object; if you return the atom `DON'T`, TEdit doesn't permit the selection; if you return the atom `CHANGED`, TEdit updates the screen. Use `CHANGED` to signal TEdit that the object has changed size or will have side effects on other parts of the screen image.

(**COPYBUTTONEVENTINFN** *IMAGEOBJ WINDOWSTREAM*)　　　　　　[IMAGEFNS Method]

　　The `COPYBUTTONEVENTINFN` method is called when you button inside an object while holding down a copy key. Many of the comments about `BUTTONEVENTINFN` apply here too. Also, see the discussion below about copying image objects between windows.

(**WHENMOVEDFN**　　*IMAGEOBJ*　　　　*TARGETWINDOWSTREAM*　　*SOURCEHOSTSTREAM*
TARGETHOSTSTREAM)　　　　　　　　　　　　　　　　[IMAGEFNS Method]

　　The `WHENMOVEDFN` method provides hooks by which the object is notified when TEdit performs an operation (`MOVEing`) on the whole object. It allows objects to have side effects.

(**WHENINSERTEDFN**　　*IMAGEOBJ*　　　　*TARGETWINDOWSTREAM*　　*SOURCEHOSTSTREAM*
TARGETHOSTSTREAM)　　　　　　　　　　　　　　　　[IMAGEFNS Method]

　　The `WHENINSERTEDFN` method provides hooks by which the object is notified when TEdit performs an operation (`INSERTing`) on the whole object. It allows objects to have side effects.

(**WHENDELETEDFN** *IMAGEOBJ TARGETWINDOWSTREAM*)　　　　　[IMAGEFNS Method]

　　The `WHENDELETEDFN` method provides hooks by which the object is notified when TEdit performs an operation (`DELETEing`) on the whole object. It allows objects to have side effects.

(**WHENCOPIEDFN**　　*IMAGEOBJ*　　　　*TARGETWINDOWSTREAM*　　*SOURCEHOSTSTREAM*
TARGETHOSTSTREAM)　　　　　　　　　　　　　　　　[IMAGEFNS Method]

　　The `WHENCOPIEDFN` method provides hooks by which the object is notified when TEdit performs an operation (`COPYing`) on the whole object. The `WHENCOPIEDFN` method is called in addition to (and after) the `COPYFN` method above. It allows objects to have side effects.

(**WHENOPERATEDONFN** *IMAGEOBJ*　　*WINDOWSTREAM*　　*HOWOPERATEDON*　　*SELECTION*
HOSTSTREAM)　　　　　　　　　　　　　　　　　[IMAGEFNS Method]

　　The `WHENOPERATEDONFN` method provides a hook for edit operations. `HOWOPERATEDON` should be one of `SELECTED`, `DESELECTED`, `HIGHLIGHTED`, and `UNHILIGHTED`. The `WHENOPERATEDONFN` differs from the `BUTTONEVENTINFN` because it is called when you extend a selection through the object. That is, the object is

treated in toto as a TEdit character. `HIGHLIGHTED` refers to the selection being highlighted on the screen, and `UNHIGHLIGHTED` means that the highlighting is being turned off.

(**PREPRINTFN** *IMAGEOBJ*)   [IMAGEFNS Method]

The `PREPRINTFN` method is called to convert the object into something that can be printed for inclusion in documents. It returns an object that the receiving window can print (using either `PRIN1` or `PRIN2`,its choice) to obtain a character representation of the object. If the `PREPRINTFN` method is `NIL`, the `OBJECTDATUM` field of *IMAGEOBJ* itself is used. TEdit uses this function when you indicate that you want to print the characters from an object rather than the object itself (presumably using `PRIN1` case).

Registering Image Objects

Each legitimate `GETFN` needs to be known to the system, to prevent various Trojan-horse problems and to allow the automatic loading of the supporting code for infrequently used `IMAGEOBJ`s. To this end, there is a global list, `IMAGEOBJGETFNS`, that contains an entry for each `GETFN`. The existence of the entry marks the `GETFN` as legitimate; the entry itself is a property list, which can hold information about the `GETFN`.

No action needs to be taken for `GETFN`s that are currently in use: the function `IMAGEFNSCREATE` automatically adds its `GETFN` argument to the list. However, packages that support obsolete versions of objects may need to explicitly add the obsolete `GETFN`s. For example, TEdit supports bit-map `IMAGEOBJ`s. Recently, a change was made in the format in which objects are stored; to retain compatibility with the old object format, there are now two `GETFN`s. The current `GETFN` is automatically on the list, courtesy of `IMAGEFNSCREATE`. However, the code file that supports the old bit-map objects contains the clause: `(ADDVARS (IMAGEOBJGETFNS (OLDGETFNNAME)))`, which adds the old `GETFN` to `IMAGEOBJGETFNS`.

For a given `GETFN`, the entry on `IMAGEOBJGETFNS` may be a property list of information. Currently the only recognized property is `FILE`.

`FILE` is the name of the file that can be loaded if the `GETFN` isn't defined. This file should define the `GETFN`, along with all the other functions needed to support that kind of `IMAGEOBJ`.

For example, the bit-map `IMAGEOBJ` implemented by TEdit use the `GETFN` `BMOBJ.GETFN2`. Its entry on `IMAGEOBJGETFNS` is `(BMOBJ.GETFN2 FILE IMAGEOBJ)`, indicating that the support code for bit-map image objects resides on the file `IMAGEOBJ`, and that the `GETFN` for them is `BMOBJ.GETFN2`.

This makes it possible to have entries for `GETFN`s whose supporting code isn't loaded—you might, for instance, have your init file add entries to `IMAGEOBJGETFNS` for the kinds of image objects you

commonly use. The system's default reading method will automatically load the code when necessary.

Reading and Writing Image Objects on Files

Image Objects can be written out to files using HPRINT and read back using HREAD. The following functions can also be used:

(**WRITEIMAGEOBJ** *IMAGEOBJ STREAM*)   [Function]

> Prints (using PRIN2) a call to READIMAGEOBJ, then calls the PUTFN for *IMAGEOBJ* to write it onto *STREAM*. During input, then, the call to READIMAGEOBJ is read and evaluated; it in turn reads back the object's description, using the appropriate GETFN.

(**READIMAGEOBJ** *STREAM GETFN NOERROR*)   [Function]

> Reads an IMAGEOBJ from *STREAM*, starting at the current file position. Uses the function *GETFN* after validating it (and loading support code, if necessary).

> If the *GETFN* can't be validated or isn't defined, READIMAGEOBJ returns an "encapsulated image object", an IMAGEOBJ that safely encapsulates all of the information in the image object. An encapsulated image object displays as a rectangle that says, "Unknown IMAGEOBJ Type" and lists the *GETFN*'s name. Selecting an encapsulated image object with the mouse causes another attempt to read the object from the file; this is so you can load any necessary support code and then get to the object.

> Warning: You cannot save an encapsulated image object on a file because there isn't enough information to allow copying the description to the new file from the old one.

> If *NOERROR* is non-NIL, READIMAGEOBJ returns NIL if it can't successfully read the object.

Copying Image Objects Between Windows

Copying between windows is implemented as follows: If a button event occurs in a window when a copy key is down, the window's COPYBUTTONEVENTFN window property is called. If this window supports copy-selection, it should track the mouse, indicating the item to be copied. When the button is released, the COPYBUTTONEVENTFN should create an image object out of the selected information, and call COPYINSERT to insert it in the current TTY window. COPYINSERT calls the COPYINSERTFN window property of the TTY window to insert this image object. Therefore, both the source and destination windows can determine how they handle copying image objects.

If the COPYBUTTONEVENTFN of a window is NIL, the BUTTONEVENTFN is called instead when a button event occurs in the window when a copy key is down, and copying from that window is not supported. If the COPYINSERTFN of the TTY window is NIL, COPYINSERT will turn the image object into a string (by calling the PREPRINTFN method of the image object) and insert it by calling BKSYSBUF.

COPYBUTTONEVENTFN   [Window Property]

> The COPYBUTTONEVENTFN of a window is called (if it exists) when a button event occurs in the window and a copy key is down. If no COPYBUTTONEVENTFN exists, the BUTTONEVENTFN is called.

COPYINSERTFN   [Window Property]

> The COPYINSERTFN of the "destination" window is called by COPYINSERT to insert something into the destination window. It is called with two arguments: the object to be inserted and the destination window. The object to be inserted can be a character string, an IMAGEOBJ, or a list of IMAGEOBJs and character strings. As a convention, the COPYINSERTFN should call BKSYSBUF if the object to be inserted insert is a character string.

(**COPYINSERT** *IMAGEOBJ*)   [Function]

> COPYINSERT inserts *IMAGEOBJ* into the window that currently has the TTY. If the current TTY window has a COPYINSERTFN, it is called, passing it *IMAGEOBJ* and the window as arguments.
>
> If no COPYINSERTFN exists and if *IMAGEOBJ* is an image object, BKSYSBUF is called on the result of calling its PREPRINTFN on it. If *IMAGEOBJ* is not an image object, it is simply passed to BKSYSBUF . In this case, BKSYSBUF will call PRIN2 with a read table taken from the process associated with the TTY window. A window that wishes to use PRIN1 or a different read table must provide its own COPYINSERTFN to do this.

Implementation of Image Streams

Interlisp does all image creation through a set of functions and data structures for device-independent graphics, known popularly as DIG. DIG is implemented through the use of a special type of stream, known as an image stream.

An image stream, by convention, is any stream that has its IMAGEOPS field (described in detail below) set to a vector of meaningful graphical operations. Using image streams, you can write programs that draw and print on an output stream without regard to the underlying device, be it a window, a disk, or a printer.

To define a new image stream type, it is necessary to put information on the variable IMAGESTREAMTYPES:

IMAGESTREAMTYPES [Variable]

This variable describes how to create a stream for a given image stream type. The value of IMAGESTREAMTYPES is an association list, indexed by the image stream type (e.g., DISPLAY, INTERPRESS, etc.). The format of a single association list item is:

```
(IMAGETYPE
    (OPENSTREAM OPENSTREAMFN)
    (FONTCREATE FONTCREATEFN)
    (FONTSAVAILABLE FONTSAVAILABLEFN))
```

OPENSTREAMFN, FONTCREATEFN, and FONTSAVAILABLEFN are "image stream methods," device-dependent functions used to implement generic image stream operations. For Interpress image streams, the association list entry is:

```
(INTERPRESS
    (OPENSTREAM OPENIPSTREAM)
    (FONTCREATE \CREATEINTERPRESSFONT)
    (FONTSAVAILABLE \SEARCHINTERPRESSFONTS))
```

(**OPENSTREAMFN** *FILE OPTIONS*) [Image Stream Method]

FILE is the file name as it was passed to OPENIMAGESTREAM, and *OPTIONS* is the *OPTIONS* property list passed to OPENIMAGESTREAM. The result must be a stream of the appropriate image type.

(**FONTCREATEFN** *FAMILY SIZE FACE ROTATION DEVICE*) [Image Stream Method]

FAMILY is the family name for the font, e.g., MODERN. *SIZE* is the body size of the font, in printer's points. *FACE* is a three-element list describing the weight, slope, and expansion of the face desired, e.g., (MEDIUM ITALIC EXPANDED). *ROTATION* is how much the font is to be rotated from the normal orientation, in minutes of arc. For example, to print a landscape page, fonts have the rotation 5400 (90 degrees). The function's result must be a FONTDESCRIPTOR with the fields filled in appropriately.

(**FONTSAVAILABLEFN** *FAMILY SIZE FACE ROTATION DEVICE*)   [Image Stream Method]

> This function returns a list of all fonts agreeing with the *FAMILY*, *SIZE*, *FACE*, and *ROTATION* arguments; any of them may be wild-carded (i.e., equal to *, which means any value is acceptable). Each element of the list should be a quintuple of the form (*FAMILY SIZE FACE ROTATION DEVICE*).

> Where the function looks is an implementation decision: the FONTSAVAILABLEFN for the display device looks at DISPLAYFONTDIRECTORIES, the Interpress code looks on INTERPRESSFONTDIRECTORIES, and implementors of new devices should feel free to introduce new search path variables.

As indicated above, image streams use a field that no other stream uses: IMAGEOPS. IMAGEOPS is an instance of the IMAGEOPS data type and contains a vector of the stream's graphical methods. The methods contained in the IMAGEOPS object can make arbitrary use of the stream's IMAGEDATA field, which is provided for their use, and may contain any data needed.

> **IMAGETYPE**   [IMAGEOPS Field]

> Value is the name of an image type. Monochrome display streams have an IMAGETYPE of DISPLAY; color display streams are identified as (COLOR DISPLAY). The IMAGETYPE field is informational and can be set to anything you choose.

> **IMFONTCREATE**   [IMAGEOPS Field]

> Value is the device name to pass to FONTCREATE when fonts are created for the stream.

> The remaining fields are all image stream methods, whose value should be a device-dependent function that implements the generic operation. Most methods are called by a similarly-named function, e.g. the function DRAWLINE calls the IMDRAWLINE method. All coordinates that refer to points in a display device's space are measured in the device's units. (The IMSCALE method provides access to a device's scale.) For arguments that have defaults (such as the BRUSH argument of DRAWCURVE), the default is substituted for the NIL argument before it is passed to the image stream method. Therefore, image stream methods do not have to handle defaults.

(**IMCLOSEFN** *STREAM*)   [Image Stream Method]

> Called before a stream is closed with CLOSEF. This method should flush buffers, write header or trailer information, etc.

(**IMDRAWLINE** *STREAM* X_1 Y_1 X_2 Y_2 *WIDTH OPERATION COLOR DASHING*)   [Image Stream Method]

> Draws a line of width *WIDTH* from (X_1, Y_1) to (X_2, Y_2). See DRAWLINE.

(**IMDRAWCURVE** *STREAM KNOTS CLOSED BRUSH DASHING*)   [Image Stream Method]

> Draws a curve through *KNOTS*. See DRAWCURVE.

(**IMDRAWCIRCLE** *STREAM CENTERX CENTERY RADIUS BRUSH DASHING*)   [Image Stream Method]

>   Draws a circle of radius *RADIUS* around (*CENTERX, CENTERY*). See DRAWCIRCLE.

(**IMDRAWELLIPSE** *STREAM CENTERX CENTERY SEMIMINORRADIUS SEMIMAJORRADIUS ORIENTATION BRUSH DASHING*)   [Image Stream Method]

>   Draws an ellipse around (*CENTERX, CENTERY*). See DRAWELLIPSE.

(**IMFILLPOLYGON** *STREAM POINTS TEXTURE*)   [Image Stream Method]

>   Fills in the polygon outlined by *POINTS* on the image stream *STREAM*, using the texture *TEXTURE*. See FILLPOLYGON.

(**IMFILLCIRCLE** *STREAM CENTERX CENTERY RADIUS TEXTURE*)   [Image Stream Method]

>   Draws a circle filled with texture *TEXTURE* around (*CENTERX, CENTERY*). See FILLCIRCLE.

(**IMBLTSHADE** *TEXTURE STREAM DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT OPERATION CLIPPINGREGION*)   [Image Stream Method]

>   The texture-source case of BITBLT. *DESTINATIONLEFT*, *DESTINATIONBOTTOM*, *WIDTH, HEIGHT*, and *CLIPPINGREGION* are measured in *STREAM*'s units. This method is invoked by the functions BITBLT and BLTSHADE.

(**IMBITBLT** *SOURCEBITMAP SOURCELEFT SOURCEBOTTOM STREAM DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT SOURCETYPE OPERATION TEXTURE CLIPPINGREGION CLIPPEDSOURCELEFT CLIPPEDSOURCEBOTTOM*) *SCALE* [Image Stream Method]

>   Contains the bit-map-source cases of BITBLT. *SOURCELEFT, SOURCEBOTTOM, CLIPPEDSOURCELEFT, CLIPPEDSOURCEBOTTOM, WIDTH*, and *HEIGHT* are measured in pixels; *DESTINATIONLEFT*, *DESTINATIONBOTTOM*, and *CLIPPINGREGION* are in the units of the destination stream.

(**IMSCALEDBITBLT** *SOURCEBITMAP SOURCELEFT SOURCEBOTTOM STREAM DESTINATIONLEFT DESTINATIONBOTTOM WIDTH HEIGHT SOURCETYPE OPERATION TEXTURE CLIPPINGREGION CLIPPEDSOURCELEFT CLIPPEDSOURCEBOTTOM SCALE*) [Image Stream Method]

>   A scaled version of IMBITBLT. Each pixel in *SOURCEBITMAP* is replicated *SCALE* times in the X and Y directions; currently, *SCALE* must be an integer.

(**IMMOVETO** *STREAM X Y*)   [Image Stream Method]

>   Moves to (*X,Y*). This method is invoked by the function MOVETO. If IMMOVETO is not supplied, a default method composed of calls to the IMXPOSITION and IMYPOSITION methods is used.

(**IMSTRINGWIDTH** *STREAM STR RDTBL*)   [Image Stream Method]

> Returns the width of string STR in *STREAM*'s units, using *STREAM*'s current font. This is envoked when STRINGWIDTH is passed a stream as its FONT argument. If IMSTRINGWIDTH is not supplied, it defaults to calling STRINGWIDTH on the default font of *STREAM*.

(**IMCHARWIDTH** *STREAM CHARCODE*)   [Image Stream Method]

> Returns the width of character *CHARCODE* in *STREAM*'s units, using *STREAM*'s current font. This is invoked when CHARWIDTH is passed a stream as its FONT argument. If IMCHARWIDTH is not supplied, it defaults to calling CHARWIDTH on the default font of *STREAM*.

(**IMCHARWIDTHY** *STREAM CHARCODE*)   [Image Stream Method]

> Returns the Y componant of the width of character *CHARCODE* in *STREAM*'s units, using *STREAM*'s current font. This is envoked when CHARWIDTHY is passed a stream as its FONT argument. If IMCHARWIDTHY is not supplied, it defaults to calling CHARWIDTHY on the default font of *STREAM*.

(**IMBITMAPSIZE** *STREAM BITMAP DIMENSION*)   [Image Stream Method]

> Returns the size that *BITMAP* will be when BITBLTed to *STREAM*, in *STREAM*'s units. *DIMENSION* can be one of *WIDTH, HEIGHT*, or NIL, in which case the dotted pair (*WIDTH . HEIGHT*) will be returned.

> This is envoked by BITMAPIMAGESIZE. If IMBITMAPSIZE is not supplied, it defaults to a method that multiplies the bitmap height and width by the scale of *STREAM*.

(**IMNEWPAGE** *STREAM*)   [Image Stream Method]

> Causes a new page to be started. The X position is set to the left margin, and the Y position is set to the top margin plus the linefeed. If not supplied, defaults to (\OUTCHAR STREAM (CHARCODE ^L)). Envoked by DSPNEWPAGE.

(**IMTERPRI** *STREAM*)   [Image Stream Method]

> Causes a new line to be started. The X position is set to the left margin, and the Y position is set to the current Y position plus the linefeed. If not supplied, defaults to (\OUTCHAR STREAM (CHARCODE EOL)). Envoked by TERPRI.

(**IMRESET** *STREAM*)   [Image Stream Method]

> Resets the X and Y position of *STREAM*. The X coordinate is set to its left margin; the Y coordinate is set to the top of the clipping region minus the font ascent. Envoked by DSPRESET.

The following methods all have corresponding `DSPxx` functions (e.g., `IMYPOSITION` corresponds to `DSPYPOSITION`) that invoke them. They also have the property of returning their previous value; when called with `NIL` they return the old value without changing it.

(**IMCLIPPINGREGION** *STREAM REGION*)   [Image Stream Method]

    Sets a new clipping region on *STREAM*.

(**IMXPOSITION** *STREAM XPOSITION*)   [Image Stream Method]

    Sets the X-position on *STREAM*.

(**IMYPOSITION** *STREAM YPOSITION*)   [Image Stream Method]

    Sets a new Y-position on *STREAM*.

(**IMFONT** *STREAM FONT*)   [Image Stream Method]

    Sets *STREAM*'s font to be *FONT*.

(**IMLEFTMARGIN** *STREAM LEFTMARGIN*)   [Image Stream Method]

    Sets *STREAM*'s left margin to be *LEFTMARGIN*. The left margin is defined as the X-position set after the new line.

(**IMRIGHTMARGIN** *STREAM RIGHTMARGIN*)   [Image Stream Method]

    Sets *STREAM*'s right margin to be *RIGHTMARGIN*. The right margin is defined as the maximum X-position at which characters are printed; printing beyond it causes a new line.

(**IMTOPMARGIN** *STREAM YPOSITION*)   [Image Stream Method]

    Sets *STREAM*'s top margin (the Y-position of the tops of characters that is set after a new page) to be *YPOSITION*.

(**IMBOTTOMMARGIN** *STREAM YPOSITION*)   [Image Stream Method]

    Sets *STREAM*'s bottom margin (the Y-position beyond which any printing causes a new page) to be *YPOSITION*.

(**IMLINEFEED** *STREAM DELTA*)   [Image Stream Method]

    Sets *STREAM*'s line feed distance (distance to move vertically after a new line) to be *DELTA*.

(**IMSCALE** *STREAM SCALE*)   [Image Stream Method]

    Returns the number of device points per screen point (a screen point being ~1/72 inch). *SCALE* is ignored.

(**IMSPACEFACTOR** *STREAM FACTOR*)   [Image Stream Method]

> Sets the amount by which to multiply the natural width of all following space characters on *STREAM*; this can be used for the justification of text. The default value is 1. For example, if the natural width of a space in *STREAM*'s current font is 12 units, and the space factor is set to two, spaces appear 24 units wide. The values returned by STRINGWIDTH and CHARWIDTH are also affected.

(**IMOPERATION** *STREAM OPERATION*)   [Image Stream Method]

> Sets the default BITBLT *OPERATION* argument.

(**IMBACKCOLOR** *STREAM COLOR*)   [Image Stream Method]

> Sets the background color of *STREAM*.

(**IMCOLOR** *STREAM COLOR*)   [Image Stream Method]

> Sets the default color of *STREAM*.

In addition to the IMAGEOPS methods described above, there are two other important methods, which are contained in the stream itself. These fields can be installed using a form like (replace (STREAM OUTCHARFN) of STREAM with (FUNCTION MYOUTCHARFN)). Note: You need to have loaded the Interlisp-D system declarations to manipulate the fields of STREAMs. The declarations can be loaded by loading the Lisp Library package SYSEDIT.

(**STRMBOUTFN** *STREAM CHARCODE*)   [Stream Method]

> The function called by BOUT.

(**OUTCHARFN** *STREAM CHARCODE*)   [Stream Method]

> The function that is called to output a single byte. This is like STRMBOUTFN, except for being one level higher: it is intended for text output. Hence, this function should convert (*CHARCODE* EOL) into the stream's actual end-of-line sequence and should adjust the stream's CHARPOSITION appropriately before invoking the stream's STRMBOUTFN (by calling BOUT) to actually put the character. Defaults to \FILEOUTCHARFN, which is probably incorrect for an image stream.

27. WINDOWS AND MENUS

Windows provide a means by which different programs can share a single display harmoniously. Rather than having every program directly manipulating the screen bitmap, all display input/output operations are directed towards windows, which appear as rectangular regions of the screen, with borders and titles. The Interlisp-D window system provides both interactive and programmatic constructs for creating, moving, reshaping, overlapping, and destroying windows in such a way that a program can use a window in a relatively transparent fashion (see the Windows section below). This allows existing Interlisp programs to be used without change, while providing a base for experimentation with more complex windows in new applications.

Menus are a special type of window provided by the window system, used for displaying a set of items to the user, and having the user select one using the mouse and cursor. The window system uses menus to provide the interactive interface for manipulating windows. The menu facility also allows users to create and use menus in interactive programs (see the Menus section below).

Sometimes, a program needs to use a number of windows, displaying related information. The attached window facility (see the Attached Windows section below) makes it easy to manipulate a group of windows as a single unit, moving and reshaping them together.

This chapter documents the Interlisp-D window system. First, it describes the default windows and menus supplied by the window system. Then, the programmatic facilities for creating windows. Next, the functions for using menus. Finally, the attached window facility.

Warning: The window system assumes that all programs follow certain conventions concerning control of the screen. All user programs should use perform display operations using windows and menus. In particular, user programs should not perform operate directly on the screen bitmap; otherwise the window system will not work correctly. For specialized applications that require taking complete control of the display, the window system can be turned off (and back on again) with the following function:

(**WINDOWWORLD** *FLAG*)   [NoSpread Function]

The window system is turned on if *FLAG* is T and off if *FLAG* is NIL. WINDOWWORLD returns the previous state of the window system (T or NIL). If WINDOWWORLD is given no arguments, it simply returns the current state without affecting the window system.

Using the Window System

When Medley is initially started, the display screen lights up, showing a number of windows, including the following:

This window is the "logo window," used to identify the system. The logo window is bound to the variable LOGOW until it is closed. The user can create other windows like this by calling the following function:

(**LOGOW** *STRING WHERE TITLE ANGLEDELTA*)　　　　　　　　　　　　　　　[Function]

> Creates a window formatted like the "logo window." *STRING* is the string to be printed in big type in the window; if NIL, "Medley" is used. *WHERE* is the position of the lower-left corner of the window; if NIL, the user is asked to specify a position. *TITLE* is the window title to use; if NIL, it defaults to the Xerox copyright notice and date. *ANGLEDELTA* specifies the angle (in degrees) between the boxes in the picture; if NIL, it defaults to 23 degrees.

This window is the "executive window," used for typing expressions and commands to the Interlisp-D executive, and for the executive to print any results (see Chapter 13). For example, in the above picture, the user typed in (PLUS 3 4), the executive evaluated it, and printed out the result, 7. The upward-pointing arrow (⋏) is the flashing caret, which indicates where the next keyboard typein will be printed (see the TTY Process and the Caret section in this chapter).

This window is the "prompt window," used for printing various system prompt messages. It is available to user programs through the following functions:

PROMPTWINDOW　　　　　　　　　　　　　　　　　　　　　　　　　[Variable]

> Global variable containing the prompt window.

(**PROMPTPRINT** *EXP_1 ... EXP_N*)　　　　　　　　　　　　　　　[NoSpread Function]

> Clears the prompt window, and prints *EXP_1* through *EXP_N* in the prompt window.

(**CLRPROMPT**)   [Function]

>   Clears the prompt window.

The Medley window system allows the user to interactively manipulate the windows on the screen, moving them around, changing their shape, etc. by selecting various operations from a menu.

For most windows, pressing the RIGHT mouse button when the cursor is inside a window during I/O wait will cause the window to come to the top and a menu of window operations to appear.

```
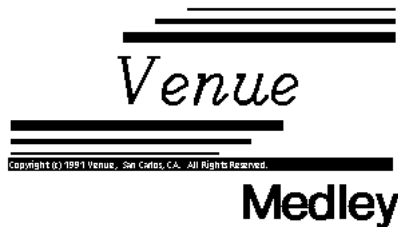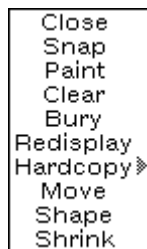Close
Snap
Paint
Clear
Bury
Redisplay
Hardcopy▶
Move
Shape
Shrink
```

If a command is selected from this menu (by releasing the right mouse key while the cursor is over a command), the selected operation will be applied to the window in which the menu was brought up. It is possible for an applications program to redefine the action of the RIGHT mouse button. In these cases, there is a convention that the default command menu may be brought up by depressing the RIGHT button when the cursor is in the header or border of a window (see the Mouse Activity in Windows section in this chapter). The operations are:

Close   [Window Menu Command]

>   Closes the window, i.e, removes it from the screen. (See CLOSEW in the Opening and Closing Windows section in this chapter.)

Snap   [Window Menu Command]

>   Prompts for a region on the screen and makes a new window whose bits are a snapshot of the bits currently in that region. Useful for saving some particularly choice image before the window image changes.

Paint   [Window Menu Command]

>   Switches to a mode in which the cursor can be used like a paint brush to draw in a window. This is useful for making notes on a window. While the LEFT button is down, bits are added. While the MIDDLE button is down, they are erased. The RIGHT button pops up a command menu that allows changing of the brush shape, size and shade, changing the mode of combining the brush with the existing bits, or stopping paint mode.

Clear   [Window Menu Command]

>   Clears the window and repositions it to the left margin of the first line of text (below the upper left corner of the window by the amount of the font ascent).

Bury [Window Menu Command]

Puts the window on the bottom of the occlusion stack, thereby exposing any windows that it was hiding.

Redisplay [Window Menu Command]

Redisplays the window. (See REDISPLAYW in the Redisplaying Windows section in this chapter.)

Hardcopy [Window Menu Command]

Prints the contents of the window to the printer. If the window has a window property HARDCOPYFN, it is called with two arguments, the window and an image stream to print to, and the HARDCOPYFN must do the printing. In this way, special windows can be set up that know how to print their contents in a particular way. If the window does not have a HARDCOPYFN, the bitmap image of the window (including the border and title) are printed on the file or printer.

To save the image in a Press or Interpress-format file, or to send it to a non-default printer, use the submenu of the Hardcopy command, indicated by a gray triangle on the right edge of the Hardcopy menu item. If the mouse is moved off of the right of the menu item, another pop-up menu will apear giving the choices "To a file" or "To a printer." If "To a file" is selected, the user is prompted to supply a file name, and the format of the file (Press, Interpress, etc.), and the specified region will be stored in the file.

If "To a printer" is selected, the user is prompted to select a printer from the list of known printers, or to type the name of another printer. If the printer selected is not the first printer on DEFAULTPRINTINGHOST (see Chapter 29), the user will be asked whether to move or add the printer to the beginning of this list, so that future printing will go to the new printer.

Move [Window Menu Command]

Moves the window to a location specified by pressing and then releasing the LEFT button. During this time a ghost frame will indicate where the window will reappear when the key is released. (See GETBOXPOSITION in the Interactive Display Functions section below.)

Shape [Window Menu Command]

Allows the user to specify a new region for the existing window contents. If the LEFT button is used to specify the new region, the reshaped window can be placed anywhere. If the MIDDLE button is used, the cursor will start out tugging at the nearest corner of the existing window, which is useful for making small adjustments in a window that is already positioned correctly. This is done by calling the function SHAPEW (see the Reshaping Windows section below).

Occasionally, a user will have a number of large windows on the screen, making it difficult to access those windows being used. To help with the problem of screen space management, the Interlisp-D window system allows the creation of "icons." An icon is a small rectangle (containing text or a

bitmap) which is a "shrunken-down" form of a particular window. Using the Shrink and Expand commands, the user can shrink windows not currently being used into icons, and quickly restore the original windows at any time.

Shrink [Window Menu Command]

> Removes the window from the screen and brings up its icon. (See SHRINKW in the Shrinking Windows into Icons section in this chapter) The window can be restored by selecting Expand from the window command menu of the icon.

If the RIGHT button is pressed while the cursor is in an icon, the window command menu will contain a slightly different set of commands. The Redisplay and Clear commands are removed, and the Shrink command is replaced with the Expand command:

Expand [Window Menu Command]

> Restores the window associated with this icon and removes the icon. (See EXPANDW in the Shrinking Windows into Icons section in this chapter.)

> If the RIGHT button is pressed while the cursor is not in any window, a "background menu" appears with the following operations:

Idle [Background Menu Command]

> Enters "idle mode" (see Chapter 12), which blacks out the display screen to save the phosphor. Idle mode can be exited by pressing any key on the keyboard or mouse. This menu command has subitems that allow the user to interactively set idle options to erase the password cache (for security), to request a password before exiting idle mode, to change the timeout before idle mode is entered automatically, etc.

SaveVM [Background Menu Command]

> Calls the function SAVEVM (see Chapter 12), which writes out all of the dirty pages of the virtual memory. After a SAVEVM, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the SAVEVM) should you experience a system crash or other disaster.

Snap [Background Menu Command]

> The same as the window menu command Snap described above.

Hardcopy [Background Menu Command]

> Prompts for a region on the screen, and sends the bitmap image to the printer by calling HARDCOPYW (see Chapter 29). Note that the region can cross window boundaries.

> Like the Hardcopy window menu command (above), the user can print to a file or specify a printer by using a submenu.

PSW [Background Menu Command]

> Prompts the user for a position on the screen, and creates a "process status window" that allows the user to examine and manipulate all of the existing processes (see Chapter 23).

Various system utilities (TEdit, SEdit, `TTYIN`) allow information to be "copy-inserted" at the current cursor position by selecting it with the "copy" key held down (Normally the shift keys are the "copy" key; this action can be changed in the key action table.) To "copy-insert" the bitmap of a snap into a Tedit document. If the right mouse button is pressed in the background with the copy key held down, a menu with the single item "`SNAP`" appears. If this item is selected, the user is prompted to select a region, and a bitmap containing the bits in that region of the screen is inserted into the current tty process, if that process is able to accept image objects.

Some built-in facilities and Lispusers packages add commands to the background menu, to provide an easy way of calling the different facilities. The user can determine what these new commands do by holding the `RIGHT` button down for a few seconds over the item in question; an explanatory message will be printed in the prompt window.

Changing the Window System

The following functions provide a functional interface to the interactive window operations so that user programs can call them directly.

(**DOWINDOWCOM** *WINDOW*)   [Function]

If *WINDOW* is a *WINDOW* that has a `DOWINDOWCOMFN` window property, it `APPLY`s that property to *WINDOW*. Shrunken windows have a `DOWINDOWCOMFN` property that presents a window command menu that contains "expand" instead of "shrink".

If *WINDOW* is a *WINDOW* that doesn't have a `DOWINDOWCOMFN` window property, it brings up the window command menu. The initial items in these menus are described above. If the user selects one of the items from the provided menu, that item is `APPLY`ed to *WINDOW*.

If *WINDOW* is `NIL`, `DOBACKGROUNDCOM` (below) is called.

If *WINDOW* is not a *WINDOW* or `NIL`, `DOWINDOWCOM` simply returns without doing anything.

(**DOBACKGROUNDCOM**)   [Function]

Brings up the background menu. The initial items in this menu are described above. If the user selects one of the items from the menu, that item is `EVAL`ed.

The window command menu for unshrunken windows is cached in the variable WindowMenu. To change the entries in this menu, the user should change the change the menu "command lists" in the variable WindowMenuCommands, and set the appropriate menu variable to a non-`MENU`, so the menu will be recreated. This provides a way of adding commands to the menu, of changing its font or of restoring the menu if it gets clobbered. The window command menus for icons and the background have similar pairs of variables, documented below. The "command lists" are in the format of the `ITEMS` field of a menu (see the Menu Fields section below), except as specified below.

Note: Command menus are recreated using the current value of `MENUFONT`.

WindowMenu [Variable]
WindowMenuCommands [Variable]

> The menu that is brought up in response to a right button in an unshrunken window is stored on the variable `WindowMenu`. If `WindowMenu` is set to a non-`MENU`, the menu will be recreated from the list of commands `WindowMenuCommands`. The `CADR` of each command added to `WindowMenuCommands` should be a function name that will be `APPLY`ed to the window.

IconWindowMenu [Variable]
IconWindowMenuCommands [Variable]

> The menu that is brought up in response to a right button in a shrunken window is stored on the variable `IconWindowMenu`. If it is `NIL`, it is recreated from the list of commands `IconWindowMenuCommands`. The `CADR` of each command added a function name that will be `APPLY`ed to the window.

BackgroundMenu [Variable]
BackgroundMenuCommands [Variable]

> The menu that is brought up in response to a right button in the background is stored on the variable `BackgroundMenu`. If it is `NIL`, it is recreated from the list of commands BackgroundMenuCommands. The `CADR` of each command added to BackgroundMenuCommands should be a form that will be `EVAL`ed.

BackgroundCopyMenu [Variable]
BackgroundCopyMenuCommands [Variable]

> The menu that is brought up in response to a right button in the background when the copy key is down is stored on the variable `BackgroundCopyMenu`. If it is `NIL`, it is recreated from the list of commands BackgroundCopyMenuCommands. The `CADR` of each command added to `BackgroundCopyMenuCommands` should be a form that will be `EVAL`ed.

Interactive Display Functions

The following functions can be used by programs to allow the user to interactively specify positions or regions on the display screen.

(**GETPOSITION** *WINDOW CURSOR*) [Function]

> Returns a `POSITION` that is specified by the user. `GETPOSITION` waits for the user to press and release the left button of the mouse and returns the cursor position at the time of release. If *WINDOW* is a *WINDOW*, the position will be in the coordinate system of *WINDOW*'s display stream. If *WINDOW* is `NIL`, the position will be in screen coordinates. If *CURSOR* is a *CURSOR* (see Chapter 30), the cursor will be changed to it while `GETPOSITION` is running. If *CURSOR* is `NIL`, the value of the system variable CROSSHAIRS will be used as the cursor: ⊕.

(**GETBOXPOSITION** *BOXWIDTH BOXHEIGHT ORGX ORGY WINDOW PROMPTMSG*)   [Function]

Allows the user to position a "ghost" region of size *BOXWIDTH* by *BOXHEIGHT* on the screen, and returns the POSITION of the lower left corner of the region. If *PROMPTMSG* is non-NIL, GETBOXPOSITION first prints it in the PROMPTWINDOW. GETBOXPOSITION then changes the cursor to a box (using the global variable BOXCURSOR: ■). If *ORGX* and *ORGY* are numbers, they are taken to be the original position of the region, and the cursor is moved to the nearest corner of that region. A ghost region is locked to the cursor so that if the cursor is moved, the ghost region moves with it. If *ORGX* and *ORGY* are numbers, the corner of the region formed by (*ORGX ORGY BOXWIDTH BOXHEIGHT*) that is nearest the cursor position is locked, otherwise the lower left corner is locked. The user can change to another corner by holding down the right button. With the right button down, the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the mouse will snap to the nearest corner, which will then become locked to the cursor. (The held corner can be changed after the left or middle button is down by holding both the original button and the right button down while the cursor is moved to the desired new corner, then letting up just the right button.) When the left or middle button is pressed and released, the lower left corner of the region at the time of release is returned. If *WINDOW* is a *WINDOW*, the returned position will be in *WINDOW*'s coordinate system; otherwise it will be in screen coordinates.

Example:

```
(GETBOXPOSITION 100 200 NIL NIL NIL
        "Specify the position of the command area.")
```

prompts the user for a 100 wide by 200 high region and returns its lower left corner in screen coordinates.

(**GETREGION** *MINWIDTH MINHEIGHT OLDREGION NEWREGIONFN NEWREGIONFNARG INITCORNERS*)   [Function]

Lets the user specify a new region and returns that region in screen coordinates. GETREGION prompts for a region by displaying a four-pronged box next to the cursor arrow at one corner of a "ghost" region: ⬉. If the user presses the left button, the corner of a "ghost" region opposite the cursor is locked where it is. Once one corner has been fixed, the ghost region expands as the cursor moves.

To specify a region:

1. Move the ghost box so that the corner opposite the cursor is at one corner of the intended region.

2. Press the left button.

3. Move the cursor to the position of the opposite corner of the intended region while holding down the left button.

4. Release the left button.

Before one corner has been fixed, one can switch the cursor to another corner of the ghost region by holding down the right button. With the right button down, the cursor changes to a "forceps" (⚷) and the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the cursor will snap to the nearest corner of the ghost region.

After one corner has been fixed, one can still switch to another corner. To change to another corner, continue to hold down the left button and hold down the right button also. With both buttons down, the cursor can be moved across the screen without effect on the ghost region frame. When the right button is released, the cursor will snap to the nearest corner, which will become the moving corner. In this way, the region may be moved all over the screen, before its size and position is finalized.

The size of the initial ghost region is controlled by the *MINWIDTH*, *MINHEIGHT*, *OLDREGION*, and *INITCORNERS* arguments.

If *INITCORNERS* is non-NIL, it should be a list specifying the initial corners of a ghost region of the form (*BASEX BASEY OPPX OPPY*), where (*BASEX, BASEY*) describes the anchored corner of the box, and (*OPPX, OPPY*) describes the trackable corner (in screen coordinates). The cursor is moved to (*OPPX, OPPY*).

If *INITCORNERS* is NIL, the ghost region will be *MINWIDTH* wide and *MINHEIGHT* high. If *MINWIDTH* or *MINHEIGHT* is NIL, 0 is used. Thus, for a call to GETREGION with no arguments specified, there will be no initial ghost region. The cursor will be in the lower right corner of the region, if there is one.

If *OLDREGION* is a region and the user presses the middle button, the corner of *OLDREGION* farthest from the cursor position is fixed and the corner nearest the cursor is locked to the cursor.

MINWIDTH and *MINHEIGHT*, if given, are the smallest *WIDTH* and *HEIGHT* that the returned region will have. The ghost image will not get any smaller than *MINWIDTH* by *MINHEIGHT*.

If *NEWREGIONFN* is non-NIL, it will be called to determine values for the positions of the corners. This provides a way of "filtering" prospective regions; for instance, by restricting the region to lie on an arbitrary grid. When the user is specifying a region, the region is determined by two of its corners, one that is fixed and one that is tracking the cursor. Each time the cursor moves or a mouse button is pressed, *NEWREGIONFN* is called with three arguments: *FIXEDPOINT*, the position of the fixed corner of the prospective region; *MOVINGPOINT*, the position of the opposite corner of the prospective region; and *NEWREGIONFNARG*. *NEWREGIONFNARG* allows the caller of GETREGION to pass information to the *NEWREGIONFN*.

The first time a button is pressed and when the user changes the moving corner via right buttoning, *MOVINGPOINT* is NIL and *FIXEDPOINT* is the position the user selected for the fixed corner of the new region. In this case, the position returned by *NEWREGIONFN* will be used for the fixed corner instead of the one proposed by the user. For all other calls, *FIXEDPOINT* is the position of the fixed corner (as returned by the previous call) and

MOVINGPOINT is the new position the user selected for the opposite corner. In these cases, the value of *NEWREGIONFN* is used for the opposite corner instead of the one proposed by the user. In all cases, the ghost region is drawn with the values returned by *NEWREGIONFN*. *NEWREGIONFN* can be a list of functions in which case they are called in order with each being passed the result of calling the previous and the value of the last one used as the point.

(**GETBOXREGION** *WIDTH HEIGHT ORGX ORGY WINDOW PROMPTMSG*)   [Function]

Performs the same prompting as GETBOXPOSITION and returns the REGION specified by the user instead of the POSITION of its lower left corner.

(**MOUSECONFIRM** *PROMPTSTRING HELPSTRING WINDOW DON'TCLEARWINDOWFLG*)
 [Function]

MOUSECONFIRM provides a simple way for the user to confirm or abort some action simply by using the mouse buttons. It prints the strings *PROMPTSTRING* and

HELPSTRING in the window *WINDOW*, changes the cursor to a "little mouse" cursor: (stored in the variable MOUSECONFIRMCURSOR), and waits for the user to press the left button to confirm, or any other button to abort. If the left button was the last button released, returns T, else NIL.

If *PROMPTSTRING* is NIL, it is not printed out. If *HELPSTRING* is NIL, the string "Click LEFT to confirm, RIGHT to abort." is used. If *WINDOW* is NIL, the prompt window is used.

Normally, MOUSECONFIRM clears *WINDOW* before returning. If *DON'TCLEARWINDOWFLG* is non-NIL, the window is not cleared.

Windows

A window specifies a region of the screen, a display stream, functions that get called when the window undergoes certain actions, and various other items of information. The basic model is that a window is a passive collection of bits (on the screen). On top of this basic level, the system supports many different types of windows that are linked to the data structures displayed in them and provide selection and redisplaying routines. In addition, it is possible for the user to create new types of windows by providing selection and displaying functions for them.

Windows are ordered in depth from user to background. Windows in front of others obscure the latter. Operating on a window generally brings it to the top.

Windows are located at a certain position on the screen. Each window has a clipping region that confines all bits written to it to a region that allows a border around the window, and a title above it.

Each window has a display stream associated with it (see Chapter 27), and either a window or its display stream can be passed interchangeably to all system functions. There are dependencies

between the window and its display stream that the user should not disturb. For instance, the destination bitmap of the display stream of a window must always be the screen bitmap. The X offset, Y offset, and Clipping Region fields of the display stream should not be changed.

Windows can be created by the user interactively, under program control, or may be created automatically by the system.

Windows are in one of two states: "open" or "closed". In an "open" state, a window is visible on the screen (unless it is covered by other open windows or off the edge of the screen) and accessible to mouse operations. In a "closed" state, a window is not visible and not accessible to mouse operations. Any attempt to print or draw on a closed window will open it.

Window Properties

The behavior of a window is controlled by a set of "window properties." Some of these are used by the system. However, any arbitrary property name may be used by a user program to associate information with a window. For many applications the user will associate the structure being displayed with its window using a property. The following functions provide for reading and setting window properties:

> (**WINDOWPROP** *WINDOW PROP NEWVALUE*)   [NoSpread Function]
>
> > Returns the previous value of *WINDOW*'s *PROP* aspect. If *NEWVALUE* is given, (even if given as NIL), it is stored as the new *PROP* aspect. Some aspects cannot be set by the user and will generate errors. Any *PROP* name that is not recognized is stored on a property list associated with the window.

> (**WINDOWADDPROP** *WINDOW PROP ITEMTOADD FIRSTFLG*)   [Function]
>
> > WINDOWADDPROP adds a new item to a window property. If *ITEMTOADD* is EQ to an element of the *PROP* property of the window WINDOW, nothing is added. If the current property is not a list, it is made a list before *ITEMTOADD* added. WINDOWADDPROP returns the previous property. If *FIRSTFLG* is non-NIL, the new item goes on the front of the list; otherwise, it goes on the end of the list. If *FIRSTFLG* is non-NIL and *ITEMTOADD* is already on the list, it is moved to the front.
>
> > Many window properties (*OPENFN,   CLOSEFN*, etc.) can be a list of functions. WINDOWADDPROP is useful for adding additional functions to a window property without affecting any existing functions. Note that if the order of items in a window property is important, the list can be modified using WINDOWPROP.

> (**WINDOWDELPROP** *WINDOW PROP ITEMTODELETE*)   [Function]
>
> > WINDOWDELPROP deletes *ITEMTODELETE* from the window property *PROP* of *WINDOW* and returns the previous list if *ITEMTODELETE* was an element. If *ITEMTODELETE* was not a member of window property *PROP*, NIL is returned.

Creating Windows

(**CREATEW** *REGION TITLE BORDERSIZE NOOPENFLG*)　　　　　　　　　　[Function]

Creates a new window. *REGION* indicates where and how large the window should be by specifying the exterior region of the window. The usable height and width of the resulting window will be smaller than the height and width of the region by twice the border size and further less the height of the title, if any. If *REGION* is NIL, GETREGION is called to prompt the user for a region.

If *TITLE* is non-NIL, it is printed in the border at the top of the window. The *TITLE* is printed using the global display stream WindowTitleDisplayStream. Thus the height of the title will be (FONTPROP WindowTitleDisplayStream 'HEIGHT).

If *BORDERSIZE* is a number, it is used as the border size. If *BORDERSIZE* is not a number, the window will have a border WBorder (initially 4) bits wide.

If *NOOPENFLG* is non-NIL, the window will not be opened, i.e. displayed on the screen.

The initial X and Y positions of the window are set to the upper left corner by calling MOVETOUPPERLEFT (see Chapter 27).

(**DECODE.WINDOW.ARG** *WHERESPEC WIDTH HEIGHT TITLE BORDER NOOPENFLG*)
 [Function]

This is a useful function for creating windows. *WHERESPEC* can be a WINDOW, a REGION, a POSITION or NIL. If *WHERESPEC* is a WINDOW, it is returned. In all other cases, CREATEW is called with the arguments *TITLE BORDER* and *NOOPENFLG*. The REGION argument to CREATEW is determined from *WHERESPEC* as follows:

If *WHERESPEC* is a REGION, it is adjusted to be on the screen, then passed to CREATEW.

If *WIDTH* and *HEIGHT* are numbers and *WHERESPEC* is a POSITION, the region whose lower left corner is *WHERESPEC*, whose width is *WIDTH* and whose height is *HEIGHT* is adjusted to be on the screen, then passed to CREATEW.

If *WIDTH* and *HEIGHT* are numbers and *WHERESPEC* is not a POSITION, then GETBOXREGION is called to prompt the user for the position of a region that is WIDTH by *HEIGHT*.

If *WIDTH* and *HEIGHT* are not numbers, CREATEW is given NIL as a REGION argument.

If *WIDTH* and *HEIGHT* are used, they are used as interior dimensions for the **window.**

(**WINDOWP** *X*)　　　　　　　　　　　　　　　　　　　　　　　　　[Function]

Returns *X* if *X* is a window, NIL otherwise.

Opening and Closing Windows

(**OPENWP** *WINDOW*)　　　　　　　　　　　　　　　　　　　　　　[Function]

Returns *WINDOW*, if *WINDOW* is an open window (has not been closed); NIL otherwise.

(**OPENWINDOWS**)   [Function]

> Returns a list of all open windows.

(**OPENW** *WINDOW*)   [Function]

> If *WINDOW* is a closed window, OPENW calls the function or functions on the window
> property OPENFN of *WINDOW*, if any. If one of the OPENFNs is the atom DON'T, the
> window will not be opened. Otherwise the window is placed on the occlusion stack of
> windows and its contents displayed on the screen. If *WINDOW* is an open window, it
> returns NIL.

(**CLOSEW** *WINDOW*)   [Function]

> CLOSEW **calls the function or functions on** the window property CLOSEFN of
> *WINDOW*, if any. If one of the CLOSEFNs is the atom DON'T or returns the atom DON'T
> as a value, CLOSEW returns without doing anything further. Otherwise, CLOSEW
> removes *WINDOW* from the window stack and restores the bits it is obscuring. If *WINDOW*
> was closed, *WINDOW* is returned as the value. If it was not closed, (for example because its
> CLOSEFN returned the atom DON'T), NIL is returned as the value.
>
> *WINDOW* can be restored in the same place with the same contents (reopened) by calling
> OPENW or by using it as the source of a display operation.

OPENFN   [Window Property]

> The OPENFN window property can be a single function or a list of functions. If one of the
> OPENFNs is the atom DON'T, the window will not be opened. Otherwise, the OPENFNs
> are called after a window has been opened by OPENW, with the window as a single
> argument.

CLOSEFN   [Window Property]

> The CLOSEFN window property can be a single function or a list of functions that are
> called just before a window is closed by CLOSEW. The function(s) will be called with the
> window as a single argument. If any of the CLOSEFNs are the atom DON'T, or if the
> value returned by any of the CLOSEFNs is the atom DON'T, the window will not be
> closed.
>
> > Note: If the CAR of the CLOSEFN list is a LAMBDA word, it is treated as a
> >   single function.
> >
> > Note: A CLOSEFN should not call CLOSEW on its argument.

Redisplaying Windows

(**REDISPLAYW** *WINDOW REGION ALWAYSFLG*)   [Function]

> Redisplay the region *REGION* of the window *WINDOW*. If *REGION* is NIL, the entire
> window is redisplayed.

If *WINDOW* doesn't have a REPAINTFN, the action depends on the value of *ALWAYSFLG*. If *ALWAYSFLG* is NIL, *WINDOW* will not change and the message `"Window  has  no REPAINTFN.   Can't  redisplay."` will be printed in the prompt window. If *ALWAYSFLG* is non-NIL, REDISPLAYW acts as if REPAINTFN was NILL.

REPAINTFN   [Window Property]

The REPAINTFN window property can be a single function or a list of functions that are called to repaint parts of the window by REDISPLAYW. The REPAINTFNs are called with two arguments: the window and the region in the coordinates of the window's display stream of the area that should be repainted. Before the REPAINTFN is called, the clipping region of the window is set to clip all display operations to the area of interest so that the REPAINTFN can display the entire window contents and the results will be appropriately clipped.

> Note: CLEARW (see the Miscellaneous Window Functions section below) should not be used in REPAINTFNs because it resets the window's coordinate system. If a REPAINTFN wants to clear its region first, it should use DSPFILL (see Chapter 27).

Reshaping Windows

(**SHAPEW** *WINDOW NEWREGION*)   [Function]

Reshapes *WINDOW*. If the window property *RESHAPEFN* is the atom DON'T or a list that contains the atom DON'T, a message is printed in the prompt window, *WINDOW* is not changed, and NIL is returned. Otherwise, RESHAPEFN window property can be a single function or a list of functions that are called when a window is reshaped, to reformat or redisplay the window contents (see below). If the RESHAPEFN window property is NIL, RESHAPEBYREPAINTFN is the default.

If the region *NEWREGION* is NIL, it prompts for a region with GETREGION. When calling GETREGION, the function MINIMUMWINDOWSIZE is called to determine the minimum height and width of the window, the function WINDOWREGION is called to get the region passed as the OLDREGION argument, the window property NEWREGIONFN is used as the NEWREGIONFN argument and *WINDOW* as the NEWREGIONFNARG argument. If the window property INITCORNERSFN is non-NIL, it is applied to the window, and the value is passed as the INITCORNERS argument to GETREGION, to determine the initial size of the "ghost region." These window properties allow the window to specify the regions used for interactive calls to SHAPEW.

If the region *NEWREGION* is a REGION and its WIDTH or HEIGHT less than the minimums returned by calling the function MINIMUMWINDOWSIZE, they will be increased to the minimums.

If *WINDOW* has a window property DOSHAPEFN, it is called, passing it *WINDOW* and NEWREGION (or the region returned by GETREGION). If *WINDOW* does not have a DOSHAPEFN window property, the function SHAPEW1 is called to reshape the window.

DOSHAPEFNs are provided to implement window groups and few users should ever write them. They are tricky to write and must call SHAPEW1 eventually. The RESHAPEFN window property is a simpler hook into reshape operations.

(**SHAPEW1** *WINDOW REGION*)   [Function]

Changes *WINDOW*'s size and position on the screen to be *REGION*. After clearing the region on the screen, it calls the window's RESHAPEFN, if any, passing it three arguments: *WINDOW*; a bitmap that contains *WINDOW*'s previous screen image; and the region of *WINDOW*'s old image within the bitmap.

RESHAPEFN   [Window Property]

The RESHAPEFN window property can be a single function or a list of functions that are called when a window is reshaped by SHAPEW. If the RESHAPEFN is DON'T or a list containing DON'T, the window will not be reshaped. Otherwise, the function(s) are called after the window has been reshaped, its coordinate system readjusted to the new position, the title and border displayed, and the interior filled with texture. The RESHAPEFN should display any additional information needed to complete the window's image in the new position and shape. The RESHAPEFN is called with four arguments: (1) the window in its reshaped form, (2) a bitmap with the image of the old window in its old shape, and (3) the region within the bitmap that contains the window's old image, and (4) the region of the screen previously occupied by this window. This function is provided so that users can reformat window contents or whatever. RESHAPEBYREPAINTFN (below) is the default and should be useful for many windows.

NEWREGIONFN   [Window Property]

If SHAPEW calls GETREGION to prompt the user for a region, the value of the NEWREGIONFN window property is passed as the NEWREGIONFN argument to GETREGION.

INITCORNERSFN   [Window Property]

If this window property is non-NIL, it should be a function of one argument, a window, that returns a list specifying the initial corners of a "ghost region" of the form (BASEX BASEY OPPX OPPY), where (BASEX, BASEY) describes the anchored corner of the box, and (OPPX, OPPY) describes the trackable corner. If SHAPEW calls GETREGION to prompt the user for a region, this function is applied to the window, and the list returned is passed as the INITCORNERS argument to GETREGION, to specify the initial ghost region.

DOSHAPEFN   [Window Property]

If this window property is non-NIL, it is called by SHAPEW to reshape the window (instead of SHAPEW1). It is called with two arguments: the window and the new region.

(**RESHAPEBYREPAINTFN** *WINDOW OLDIMAGE IMAGEREGION OLDSCREENREGION*)
 [Function]

This the default window RESHAPEFN. *WINDOW* is a window that has been reshaped from the screen region *OLDSCREENREGION* to its new region (available via (WINDOWPROP WINDOW 'REGION)). *OLDIMAGE* is a bitmap that contains the image of the window from its previous location. *IMAGEREGION* is the region within *OLDIMAGE* that contains the old image.

RESHAPEBYREPAINTFN BITBLTs the old region contents into the new region. If the new shape is larger in either or both dimensions, the newly exposed areas are redisplayed via calls *WINDOW*'s REPAINTFN window property. RESHAPEBYREPAINTFN may call the REPAINTFN up to four times during a single reshape.

The choice of which areas of the window to remove or extend is done as follows. If *WINDOW*'s new region shares an edge with OLDSCREENREGION, that edge of the window image will remain fixed and any addition or reduction in that dimension will be performed on the opposite side. If *WINDOW* has an EXTENT property and the newly exposed window area is outside of it, any extra will be added so as to show EXTENT that was previously not visible. An exception to these rules is that the current X,Y position is kept visible, if it was visible before the reshape.

Moving Windows

(**MOVEW** *WINDOW POSorX Y*)   [Function]

Moves *WINDOW* to the position specified by *POSorX* and *Y* according to the following rules:

If *POSorX* is NIL, GETBOXPOSITION is called to read a position from the user. If *WINDOW* has a CALCULATEREGION window property, it will be called with *WINDOW* as an argument and should return a region which will be used to prompt the user with. If *WINDOW* does not have a CALCULATEREGION window property, the region of *WINDOW* is used to prompt with.

If *POSorX* is a POSITION, *POSorX* is used.

If *POSorX* and *Y* are both NUMBERP, a position is created using *POSorX* as the XCOORD and *Y* as the YCOORD.

If *POSorX* is a REGION, a position is created using its LEFT as the XCOORD and BOTTOM as the YCOORD.

If *WINDOW* is not open and *POSorX* is non-NIL, the window will be moved without being opened. Otherwise, it will be opened.

If *WINDOW* has the atom DON'T as a MOVEFN window property, the window will not be moved. If *WINDOW* has any other non-NIL value as a MOVEFN property, it should be a function or list of functions that will be called before the window is moved with the *WINDOW* and the new positon as its arguments. If it returns the atom DON'T, the window will not be moved. If it returns a position, the window will be moved to that position

instead of the new one. If there are more than one MOVEFNs, the last one to return a value is the one that determines where the window is moved to.

If *WINDOW* is moved and *WINDOW* has an AFTERMOVEFN window property, it should be a function or a list of functions that will be called after the window is moved with *WINDOW* as an argument.

MOVEW returns the new position, or NIL if the window could not be moved.

Note: If MOVEW moves any part of the window from off-screen onto the screen, that part is redisplayed (by calling REDISPLAYW).

(**RELMOVEW** *WINDOW POSITION*)   [Function]

Like MOVEW for moving windows but the *POSITION* is interpreted relative to the current position of *WINDOW*. Example: The following code moves *WINDOW* to the right one screen point.

```
(RELMOVEW WINDOW (create POSITION XCOORD ← 1 YCOORD
← 0))
```

CALCULATEREGION   [Window Property]

If MOVEW calls GETBOXPOSITION to prompt the user for a region, the CALCULATEREGION window property is called (passing the window as an argument. The CALCULATEREGION should returns a region to be used to prompt the user with. If CALCULATEREGION is NIL, the region of the window is used to prompt with.

MOVEFN   [Window Property]

If the MOVEFN is DON'T, the window will not be moved by MOVEW. Otherwise, if the MOVEFN is non-NIL, it should be a function or a list of functions that will be called before a window is moved with two arguments: the window being moved and the new position of the lower left corner in screen coordinates. If the MOVEFN returns DON'T, the window will not be moved. If the MOVEFN returns a POSITION, the window will be moved to that position. Otherwise, the window will be moved to the specified new position.

AFTERMOVEFN   [Window Property]

If non-NIL, it should be a function or a list of functions that will be called after the window is moved (by MOVEW) with the window as an argument.

Exposing and Burying Windows

(**TOTOPW** *WINDOW NOCALLTOTOPFNFLG*)   [Function]

Brings *WINDOW* to the top of the stack of overlapping windows, guaranteeing that it is entirely visible. If *WINDOW* is closed, it is opened. This is done automatically whenever a printing or drawing operation occurs to the window.

If `NOCALLTOTOPFNFLG` is `NIL`, the `TOTOPFN` of *WINDOW* is called. If *NOCALLTOTOPFNFLG* is `T`, it is not called, which allows a `TOTOPFN` to call `TOTOPW` without causing an infinite loop.

(**BURYW** *WINDOW*)   [Function]

Puts *WINDOW* on the bottom of the stack by moving all the windows that it covers in front of it.

TOTOPFN   [Window Property]

If non-`NIL`, whenever the window is brought to the top, the `TOTOPFN` is called (with the window as a single argument). This function may be used to bring a collection of windows to the top together.

If the `NOCALLTOPWFN` argument of `TOTOPW` is non-`NIL`, the `TOTOPFN` of the window is not called, which provides a way of avoiding infinite loops when using `TOTOPW` from within a `TOTOPFN`.

Shrinking Windows Into Icons

Occasionally, a user will have a number of large windows on the screen, making it difficult to access those windows being used. To help with the problem of screen space management, the Interlisp-D window system allows the creation of Icons. An icon is a small rectangle (containing text or a bitmap) which is a "shrunken-down" form of a particular window. Using the Shrink and Expand window menu commands (see the beginning of this chapter), the user can shrink windows not currently being used into icons, and quickly restore the original windows at any time. This facility is controlled by the following functions and window properties:

(**SHRINKW** *WINDOW TOWHAT ICONPOSITION EXPANDFN*)   [Function]

`SHRINKW` makes a small icon which represents *WINDOW* and removes *WINDOW* from the screen. Icons have a different window command menu that contains "EXPAND" instead of "SHRINK". The EXPAND command calls `EXPANDW` which returns the shrunken window to its original size and place. The icon can also be moved by pressing the `LEFT` button in it, or expanded by pressing the `MIDDLE` button in it.

The `SHRINKFN` property of the window *WINDOW* affects the operation of `SHRINKW`. If the `SHRINKFN` property of *WINDOW* is the atom `DON'T`, `SHRINKW` returns. Otherwise, the `SHRINKFN` property of the window is treated as a (list of) function(s) to apply to *WINDOW*; if any returns the atom `DON'T`, `SHRINKW` returns.

TOWHAT, if given, indicates the image the icon window will have. If *TOWHAT* is a string, atom or list, the icon's image will be that string (currently implemented as a title-only window with *TOWHAT* as the title.) If *TOWHAT* is a `BITMAP`, the icon's image will be a copy of the bitmap. If *TOWHAT* is a *WINDOW*, that window will be used as the icon.

If *TOWHAT* is not given (as is the case when invoked from the `SHRINK` window command), then the following apply in turn:

1. If the window has an ICONFN property, it gets called with the two arguments *WINDOW* and OLDICON, where *WINDOW* is the window being shrunk and OLDICON is the previously created icon, if any. The ICONFN should return one of the *TOWHAT* entities described above or return the OLDICON if it does not want to change it.

2. If the window has an ICON property, it is used as the value of *TOWHAT*.

3. If the window has neither an ICONFN or ICON property, the icon will be *WINDOW*'s title or, if *WINDOW* doesn't have a title, the date and time of the icon creation.

ICONPOSITION gives the position that the new icon will be on the screen. If it is NIL, the icon will be in the corner of the window furthest from the center of the screen.

In all but the default case, the icon is cached on the property ICONWINDOW of *WINDOW* so repeating SHRINKW reuses the same icon (unless overridden by the ICONFN described above). Thus to change the icon it is necessary to remove the ICONWINDOW property or call SHRINKW explicitly giving a *TOWHAT* argument.

(**EXPANDW** *ICONW*)   [Function]

Restores the window for which *ICONW* is an icon, and removes the icon from the screen. If the EXPANDFN window property of the main window is the atom DON'T, the window won't be expanded. Otherwise, the window will be restored to its original size and location and the EXPANDFN (or list of functions) will be applied to it.

SHRINKFN   [Window Property]

The SHRINKFN window property can be a single function or a list of functions that are called just before a window is shrunken by SHRINKW, with the window as a single argument. If any of the SHRINKFNs are the atom DON'T, or if the value returned by any of the SHRINKFNs is the atom DON'T, the window will not be shrunk.

EXPANDREGIONFN   [Window property]

EXPANDREGIONFN, if non-NIL, should be the function to be called (with the window as its argument) before the window is actually expanded.

The EXPANDREGIONFN must return NIL or a valid region, and must not do any window operations (e.g., redisplaying). If NIL is returned, the window is expanded normally, as if the EXPANDREGIONFN had not existed. The region returned specifies the new region for the main window only, not for the group including any of its attached windows. The window will be opened in its new shape, and any attached windows will be repositioned or rejustified appropriately. The main window must have a REPAINTFN which can repaint the entire window under these conditions.

As with expanding windows normally, the OPENFN for the main window is not called.

Also, the window is reshaped without checking for a special shape function (e.g., a `DOSHAPEFN`).

`ICONFN` [Window Property]

If `SHRINKW` is called without begin given a `TOWHAT` argument (as is the case when invoked from the `SHRINK` window command) and the window's `ICONFN` property is non-`NIL`, then it gets called with two arguments, the window being shrunk and the previously created icon, if any. The `ICONFN` should return one of the `TOWHAT` entities described above or return the previously created icon if it does not want to change it.

`ICON` [Window Property]

If `SHRINKW` is called without being given a `TOWHAT` argument, the window's `ICONFN` property is `NIL`, and the `ICON` property is non-`NIL`, then it is used as the value of `TOWHAT`.

`ICONWINDOW` [Window Property]

Whenever an icon is created, it is cached on the property `ICONWINDOW` of the window, so calling `SHRINKW` again will reuse the same icon (unless overridden by the `ICONFN`.

Thus, to change the icon it is necessary to remove the `ICONWINDOW` property or call `SHRINKW` explicitly giving a `TOWHAT` argument.

`DEFAULTICONFN` [Variable]

Changes how an icon is created when a window having no `ICONFN` is shrunk or when `SHRINKW`, with a `TOWHAT` argument of a string, is called. The value of `DEFAULTICONFN` is a function of two arguments (window text); text is either `NIL` or a string. `DEFAULTICONFN` returns an icon window.

The initial value of `DEFAULTICONFN` is `MAKETITLEBARICON`. It creates a window that is a title bar only; the title is either the text argument, the window's title, or "Icon made <date>" for titleless windows. `MAKETITLEBARICON` places the title bar at some corner of the main window.

An alternative behavior is available by setting `DEFAULTICONFN` to be `TEXTICON`. `TEXTICON` creates a titled icon window from the text or window's title.

You can now copy-select titled icons such as those used by FileBrowser, SEdit, TEdit, Sketch. The default behavior is that the icon's title is unread (via `BKSYSBUF`), but if the icon window has a `COPYFN` property, that gets called instead, with the icon window as its argument. For example, if the name displayed in an icon is really a symbol, and you want copy selection to cause the name to be unread correctly with respect to the package and read table of the exec you are copying into, you could put the following `COPYFN` property on the icon window:

```
(LAMBDA (WINDOW)
```

```
(IL:BKSYSBUF  <fetch  symbolic  name  from  window>  T ))
```

EXPANDFN   [Window Property]

> The EXPANDFN window property can be a single function or a list of functions. If one of
> the EXPANDFNs is the atom DON'T, the window will not be expanded. Otherwise, the
> EXPANDFNs are called after the window has been expanded by EXPANDW, with the
> window as a single argument.

Creating Icons with ICONW

ICONW is a group of functions available for building small windows of arbitrary shape. These
windows are principally for use as icons for shrinking windows; i.e., these functions are likely to be
invoked from within the ICONFN of a window. An icon is specified by supplying its image (a bitmap)
and a mask that specifies its shape. The mask is a bitmap of the same dimensions as the image whose
bits are on (black) in those positions considered to be in the image, and off (white) in those positions
where the background should show through. By using the mask and appropriate window functions,
ICONW maintains the illusion that the icon window is nonrectangular, even though the actual
window itself is rectangular. The illusion is not complete, of course. For example, if you try to select
what looks like the background (or an occluded window) around the icon but still within its
rectangular perimeter, the icon window itself is selected. Also, if you move a window occluded by an
icon, the icon never notices that the background changed behind it. Icons created with ICONW can
also have titles; some part of the image can be filled with text computed at the time the icon is created,
or text may be changed after creation.

Creating Icons

Two types of icons can be created with ICONW, a borderless window containing an image defined by
a mask and a window with a title.

(**ICONW** *IMAGE MASK POSITION NOOPENFLG*)   [Function]

> Creates a window at *POSITION*, or prompts for a position if *POSITION* is NIL. The
> window is borderless, and filled with *IMAGE*, as cookie-cut by *MASK*. If *MASK* is NIL, the
> image is considered rectangular (i.e., *MASK* defaults to a black bitmap of the same
> dimensions as *IMAGE*). If *NOOPENFLG* is T, the window is returned unopened.

(**TITLEDICONW** *ICON TITLE FONT POSITION NOOPENFLG JUST BREAKCHARS
OPERATION*)
 [Function]

Creates a titled icon at *POSITION*, or prompts for a position if *POSITION* is NIL. If *NOOPENFLG* is T, the window is returned unopened. The argument *ICON* is an instance of the record TITLEDICON, which specifies the icon image and mask, as with ICONW, and a region within the image to be used for displaying the title. Thus, the *ICON* argument is usually of the form

(create TITLEDICON ICON ← someIconImage

MASK ← iconMask TITLEREG ← someRegionWithinICON)

The title region is specified in coordinates relative to the icon, i.e., the lower-left corner of the image bitmap is (0, 0). The mask can be NIL if the icon is rectangular. The image should be white where it is covered by the title region. TITLEDICONW clears the region before printing on it. The title is printed into the specified region in the image, using *FONT*. If *FONT* is NIL it defaults to the value of DEFAULTICONFONT, initially Helvetica 10. The title is broken into multiple lines if necessary; TITLEDICONW attempts to place the breaks at characters that are in the list of character codes *BREAKCHARS*. *BREAKCHARS* defaults to (CHARCODE (SPACE ÿ)). In addition, line breaks are forced by any carriage returns in *TITLE*, independent of *BREAKCHARS*. *BREAKCHARS* is ignored if a long title would not otherwise fit in the specified region. For convenience, *BREAKCHARS* = FILE means the title is a file name, so break at file name field delimiters. The argument *JUST* indicates how the text should be justified relative to the region. It is an atom or list of atoms chosen from *TOP*, *BOTTOM*, *LEFT*, or *RIGHT*, which indicate the vertical positioning (flush to top or bottom) and/or horizontal positioning (flush to left edge or right). If *JUST* = NIL, the text is centered. The argument *OPERATION* is a display stream operation indicating how the title should be printed. If *OPERATION* is INVERT, then the title is printed white-on-black. The default *OPERATION* is REPLACE, meaning black-on-white. ERASE is the same as INVERT; PAINT is the same as REPLACE.

For convenience, TITLEDICONW can also be used to create icons that consist solely of a title, with no special image. If the argument *ICON* is NIL, TITLEDICONW creates a rectangular icon large enough to contain *TITLE*, with a border the same width as that on a regular window. The remaining arguments are as described above, except that a *JUST* of *TOP* or *BOTTOM* is not meaningful.

In the Medley release, TITLEDICONW can create icons with white text on a black background. To get this effect, your icon image must be black in the correct area, and you must specify the *OPERATION* argument as INVERT.

In Medley, you can copy- select the title of an icon.

Modifying Icons

(**ICONW.TITLE** *ICON TITLE*)   [Function]

Returns the current title of the window *ICON*, which must be a window returned by TITLEDICONW. In addition, if *TITLE* is non-NIL, makes *TITLE* the new title of the window and repaints it accordingly. To erase the current title, make *TITLE* a null string.

(**ICONW.SHADE** *WINDOW SHADE*)   [Function]

Returns the current shading of the window ICON, which must be a window returned by ICONW or TITLEDICONW. In addition, if *SHADE* is non-NIL, paints the texture *SHADE* on *WINDOW*. A typical use for this function is to communicate a change of state in a window that is shrunken, without reopening the window. To remove any shading, make *SHADE* be WHITESHADE.

Default Icons

When you shrink a window that has no ICONFN, the system currently creates an icon that looks like the window's title bar. You can make the system instead create titled icons by setting the global variable DEFAULTICONFN to the value TEXTICON.

(**TEXTICON** *WINDOW TEXT*)   [Function]

Creates a titled icon window for the main window *WINDOW* containing the text *TEXT*, or the window's title if *TEXT* is NIL.

DEFAULTTEXTICON   [Variable]

The value that TEXTICON passes to TITLEDICONW as its ICON argument. Initially it is NIL, which creates an unadorned rectangular window. However, you can set it to a TITLEDICON record of your choosing if you would like default icons to have a different appearance.

Coordinate Systems, Extents, And Scrolling

Note: The word "scrolling" has two distinct meanings when applied to Interlisp-D windows. This section documents the use of "scroll bars" on the left and bottom of a window to move an object displayed in the window. "Scrolling" also describes the feature where trying to print text off the bottom of a window will cause the contents to "scroll up." This second feature is controlled by the function DSPSCROLL (see Chapter 27).

One way of thinking of a window is as a "view" onto an object (e.g. a graph, a file, a picture, etc.) The object has its own natural coordinate system in terms of which its subparts are laid out. When the window is created, the X Offset and Y Offset of the window's display stream are set to map the origin of the object's coordinate system into the lower left point of the window's interior region. At the same time, the Clipping Region of the display stream is set to correspond to the interior of the window. From then on, the display stream's coordinate system is translated and its clipping region adjusted whenever the window is moved, scrolled or reshaped.

There are several distinct regions associated with a window viewing an object. First, there is a region in the window's coordinate system that contains the complete image of the object. This region (which can only be determined by application programs with knowledge of the "semantics" of the object) is stored as the EXTENT property of the window (below). Second, the clipping region of the display stream (obtainable with the function DSPCLIPPINGREGION, see Chapter 27) specifies the portion of the object that is actually visible in the window. This is set so that it corresponds to the interior of the window (not including the border or title). Finally, there is the region on the screen that specifies the total area that the window occupies, including the border and title. This region (in screen coordinates) is stored as the REGION property of the window (see the Miscellaneous Window Properties section below).

The window system supports the idea of scrolling the contents of a window. Scrolling regions are on the left and the bottom edge of each window. The LEFT button is used to indicate upward or leftward scrolling by the amount necessary to move the selected position to the top or the left edge. The RIGHT button is used to indicate downward or rightward scrolling by the amount necessary to move the top or left edge to the selected position. The MIDDLE button is used to indicate global placement of the object within the window (similar to "thumbing" a book). In the scroll region, the part of the object that is being viewed by the window is marked with a gray shade. If the whole scroll bar is thought of as the entire object, the shaded portion is the portion currently being viewed. This will only occur when the window "knows" how big the object is (see window property EXTENT, below).

When the button is released in a scroll region, the function SCROLLW is called. SCROLLW calls the scrolling function associated with the window to do the actual scrolling and provides a programmable entry to the scrolling operation.

(**SCROLLW** *WINDOW DELTAX DELTAY CONTINUOUSFLG*)   [Function]

> Calls the SCROLLFN window property of the window *WINDOW* with arguments *WINDOW*, *DELTAX*, *DELTAY* and *CONTINUOUSFLG*. See SCROLLFN window property below.

(**SCROLL.HANDLER** *WINDOW*)   [Function]

> This is the function that tracks the mouse while it is in the scroll region. It is called when the cursor leaves a window in either the left or downward direction. If N *MWINDOW* does not have a scroll region for this direction (e.g. the window has moved or reshaped since it was last scrolled), a scroll region is created that is SCROLLBARWIDTH wide. It then waits for SCROLLWAITTIME milliseconds and if the cursor is still inside the scroll region, it opens a window the size of the scroll region and changes the cursor to indicate the scrolling is taking place.

> When a button is pressed, the cursor shape is changed to indicate the type of scrolling (up, down, left, right or thumb). After the button is held for WAITBEFORESCROLLTIME milliseconds, until the button is released SCROLLW is called each WAITBETWEENSCROLLTIME milliseconds. These calls are made with the CONTINUOUSFLG argument set to T. If the button is released before WAITBEFORESCROLLTIME milliseconds, SCROLLW is called with the CONTINUOUSFLG argument set to NIL.

The arguments passed to SCROLLW depend on the mouse button. If the LEFT button is used in the vertical scroll region, DY is distance from cursor position at the time the button was released to the top of the window and DX is 0. If the RIGHT button is used, the inverse of this quantity is used for DY and 0 for DX. If the LEFT button is used in the horizontal scroll region, DX is distance from cursor position to left of the window and DY is 0. If the RIGHT button is used, the inverse of this quantity is used for DX and 0 for DY.

If the MIDDLE button is pressed, the distance argument to SCROLLW will be a FLOATP between 0.0 and 1.0 that indicates the proportion of the distance the cursor was from the left or top edge to the right or bottom edge.

> Note: The scrolling regions will not come up if the window has a SCROLLFN window property of NIL, has a non-NIL NOSCROLLBARS window property, or if its SCROLLEXTENTUSE property has certain values and its EXTENT is fully visible.

(**SCROLLBYREPAINTFN** *WINDOW DELTAX DELTAY CONTINUOUSFLG*)   [Function]

SCROLLBYREPAINTFN is the standard scrolling function which should be used as the SCROLLFN property for most scrolling windows.

This function, when used as a SCROLLFN, BITBLTs the bits that will remain visible after the scroll to their new location, fills the newly exposed area with texture, adjusts the window's coordinates and then calls the window's REPAINTFN on the newly exposed region. Thus this function will scroll any window that has a repaint function.

If *WINDOW* has an EXTENT property, SCROLLBYREPAINTFN will limit scrolling in the X and Y directions according to the value of the window property SCROLLEXTENTUSE.

If *DELTAX* or *DELTAY* is a FLOATP, SCROLLBYREPAINTFN will position the window so that its top or left edge will be positioned at that proportion of its EXTENT. If the window does not have an EXTENT, SCROLLBYREPAINTFN will do nothing.

If *CONTINUOUSFLG* is non-NIL, this indicates that the scrolling button is being held down. In this case, SCROLLBYREPAINTFN will scroll the distance of one linefeed height (as returned by DSPLINEFEED, see Chapter 27).

Scrolling is controlled by the following window properties:

EXTENT   [Window Property]

Used to limit scrolling operations. Accesses the extent region of the window. If non-NIL, the EXTENT is a region in the window's display stream that contains the complete image of the object being viewed by the window. User programs are responsible for updating the EXTENT. The functions UNIONREGIONS, EXTENDREGION, etc. (see Chapter 27) are useful for computing a new extent region.

In some situations, it is useful to define an EXTENT that only exists in one dimension. This may be done by specifying an EXTENT region with a width or height of -1.

SCROLLFN handling recognizes this situation as meaning that the negative EXTENT dimension is unknown.

SCROLLFN   [Window Property]

If the SCROLLFN property is NIL, the window will not scroll. Otherwise, it should be a function of four arguments: (1) the window being scrolled, (2) the distance to scroll in the horizontal direction (positive to right, negative to left), (3) the distance to scroll in the vertical direction (positive up, negative down), and (4) a flag which is T if the scrolling button is being held down. For more information, see SCROLL.HANDLER. For most scrolling windows, the SCROLLFN function should be SCROLLBYREPAINTFN.

NOSCROLLBARS   [Window Property]

If the NOSCROLLBARS property is non-NIL, scroll bars will not be brought up for this window. This disables mouse-driven scrolling of a window. This window can still be scrolled using SCROLLW.

SCROLLEXTENTUSE   [Window Property]

SCROLLBYREPAINTFN uses the SCROLLEXTENTUSE window property to limit how far scrolling can go in the X and Y directions. The possible values for SCROLLEXTENTUSE and their interpretations are:

NIL   This will keep the extent region visible or near visible. It will not scroll the window so that the top of the extent is below the top of the window, the bottom of the extent is more than one point above the top of the window, the left of the extent is to the right of the window and the right of the extent is to the left of the window. The EXTENT can be scrolled to just above the window to provide a way of "hiding" the contents of a window. In this mode the extent is either in the window or just of the top of the window.

T   The extent is not used to control scrolling. The user can scroll the window to anywhere. Having the EXTENT window property does all thumb scrolling to be supported so that the user can get back to the EXTENT by thumb scrolling.

LIMIT   This will keep the extent region visible. The window is only allowed to view within the extent.

+   This will keep the extent region visible or just off in the positive direction in either X or Y (i.e., the image will be either be visible or just off to the top and/or right.)

–   This will keep the extent region visible or just off in the negative direction in either X or Y (i.e., the image will be either be visible or just off to the left and/or bottom).

+–

-+   This will keep the extent region visible or just off in the window (i.e. the image will be either be visible or just off to the left, bottom, top or right).

(XBEHAVIOR . YBEHAVIOR) If the SCROLLEXTENTUSE is a list, the CAR is interpreted as the scrolling limit in the X behavior and the CDR as the scrolling limit in the Y behavior. XBEHAVIOR and YBEHAVIOR should each be one of the atoms (NIL T LIMIT + - +- -+). The interpretations of the atoms is the same as above except that NIL is equivalent to LIMIT.

Note: The NIL value of SCROLLEXTENTUSE is equivalent to (LIMIT . +).

Example: If the SCROLLEXTENTUSE window property of a window (with an extent defined) is (LIMIT . T), the window will scroll uncontrolled in the Y dimension but be limited to the extent region in the X dimension.

Mouse Activity in Windows

The following window properties allow the user to control the response to mouse activity in a window. The value of these properties, if non-NIL, should be a function that will be called (with the window as argument) when the specified event occurs.

These functions should be "self-contained", communicating with the outside world solely via their window argument, e.g., by setting window properties. In particular, these functions should not expect to access variables bound on the stack, as the stack context is formally undefined at the time these functions are called. Since the functions are invoked asynchronously, they perform any terminal input/output operations from their own window.

WINDOWENTRYFN   [Window Property]

Whenever a button goes down in the window and the process associated with the window is not the tty process, the WINDOWENTRYFN is called. The default is GIVE.TTY.PROCESS which gives the process associated with the window the tty and calls the BUTTONEVENTFN. WINDOWENTRYFN can be a list of functions and all will be called.

CURSORINFN   [Window Property]

Whenever the mouse moves into the window, the CURSORINFN is called. If CURSORINFN is a list of functions, all will be called.

CURSOROUTFN   [Window Property]

The CURSOROUTFN is called when the cursor leaves the window. If CURSOROUTFN is a list of functions, all will be called.

CURSORMOVEDFN   [Window Property]

The `CURSORMOVEDFN` is called whenever the cursor has moved and is inside the window. `CURSORMOVEDFN` can be a list of functions and all will be called. This allows a window function to implement "active" regions within itself by having its `CURSORMOVEDFN` determine if the cursor is in a region of interest, and if so, perform some action.

BUTTONEVENTFN   [Window Property]

The `BUTTONEVENTFN` is called whenever there is a change in the state (up or down) of the mouse buttons inside the window. Changes to the mouse state while the `BUTTONEVENTFN` is running will not be interpreted as new button events, and the `BUTTONEVENTFN` will not be re-invoked.

RIGHTBUTTONFN   [Window Property]

The `RIGHTBUTTONFN` is called in lieu of the standard window menu operation (`DOWINDOWCOM`) when the `RIGHT` button is depressed in a window. More specifically, the `RIGHTBUTTONFN` is called instead of the `BUTTONEVENTFN` when (`MOUSESTATE (ONLY RIGHT)`). If the `RIGHT` button is to be treated like any other key in a window, supply `RIGHTBUTTONFN` and `BUTTONEVENTFN` with the same function.

When an application program defines its own `RIGHTBUTTONFN`, there is a convention that the default `RIGHTBUTTONFN`, `DOWINDOWCOM` , may be executed by pressing the `RIGHT` button when the cursor is in the header or border of a window. User `RIGHTBUTTONFN`s are encouraged to follow this convention, by calling `DOWINDOWCOM` if the cursor is not in the interior region of the window.

BACKGROUNDBUTTONEVENTFN   [Variable]
BACKGROUNDCURSORINFN   [Variable]
BACKGROUNDCURSOROUTFN   [Variable]
BACKGROUNDCURSORMOVEDFN   [Variable]

These variables provide a way of taking action when there is cursor action and the cursor in in the background. They are interpreted like the corresponding window properties. If set to the name of a function, that function will be called, respectively, whenever the cursor is in the background and a button changes, when the cursor moves into the background from a window, when the cursor moved from the background into a window and when the cursor moves from one place in the background to another.

Terminal I/O and Page Holding

Each process has its own terminal i/o stream (accessed as the stream T, see Chapter 25). The terminal i/o stream for the current process can be changed to point to a window by using the function `TTYDISPLAYSTREAM`, so that output and echoing of type-in is directed to a window.

(**TTYDISPLAYSTREAM** *DISPLAYSTREAM*)   [Function]

Selects the display stream or window *DISPLAYSTREAM* to be the terminal output channel, and returns the previous terminal output display stream. `TTYDISPLAYSTREAM` puts

DISPLAYSTREAM into scrolling mode and calls PAGEHEIGHT with the number of lines that will fit into *DISPLAYSTREAM* given its current Font and Clipping Region. The line length of TTYDISPLAYSTREAM is computed (like any other display stream) from its Left Margin, Right Margin, and Font. If one of these fields is changed, its line length is recalculated. If one of the fields used to compute the number of lines (such as the Clipping Region or Font) changes, PAGEHEIGHT is not automatically recomputed. (TTYDISPLAYSTREAM (TTYDISPLAYSTREAM)) will cause it to be recomputed.

If the window system is active, the line buffer is saved in the old TTY window, and the line buffer is set to the one saved in the window of the new display stream, or to a newly created line buffer (if it does not have one). Caution: It is possible to move the TTYDISPLAYSTREAM to a nonvisible display stream or to a window whose current position is not in its clipping region.

(**PAGEHEIGHT** *N*) [Function]

If *N* is greater than 0, it is the number of lines of output that will be printed to TTYDISPLAYSTREAM before the page is held. A page is held before the *N*+1 line is printed to TTYDISPLAYSTREAM without intervening input if there is no terminal input waiting to be read. The output is held with the screen video reversed until a character is typed. Output holding is disabled if *N* is 0. PAGEHEIGHT returns the previous setting.

PAGEFULLFN [Window Property]

If the PAGEFULLFN window property is non-NIL, it will be called with the window as a single argument when the window is full (i.e., when enough has been printed since the last TTY interaction so that the next character printed will cause information to be scrolled off the top of the window.)

If the PAGEFULLFN window property is NIL, the system function PAGEFULLFN is called. PAGEFULLFN simply returns if there are characters in the type-in buffer for WINDOW, otherwise it inverts the window and waits for the user to type a character. PAGEFULLFN is user advisable.

> Note: The PAGEFULLFN window property is only called on windows which are the TTYDISPLAYSTREAM of some process.

TTY Process and the Caret

At any time, one process is designated as the TTY process, which is used for accepting keyboard input. The TTY process can be changed to a given process by calling GIVE.TTY.PROCESS (see Chapter 23), or by clicking the mouse in a window associated with the process. The latter mechanism is implemented with the following window property:

PROCESS [Window Property]

If the PROCESS window property is non-NIL, it should be a PROCESS and will be made the TTY process by GIVE.TTY.PROCESS (see Chapter 23), the default

WINDOWENTRYFN property (see above). This implements the mechanism by which the keyboard is associated with different processes.

The window system uses a flashing caret (▲) to indicate the position of the next window typeout. There is only one caret visible at any one time. The caret in the current TTY process is always visible; if it is hidden by another window, its window is brought to the top. An exception to this rule is that the flashing caret's window is not brought to the top if the user is buttoning or has a shift key down. This prevents the destination window (which has the tty and caret flashing) from interfering with the window one is trying to select text to copy from.

(**CARET** *NEWCARET*)   [Function]

Sets the shape that blinks at the location of the next output to the current process. *NEWCARET* should be one of the following:

a CURSOR object   If *NEWCARET* is a CURSOR object (see Chapter 30), it is used to give the new caret shape

OFF   Turns the caret off

NIL   The caret is not changed. CARET returns a CURSOR representing the current caret

T   Reset the caret to the value of DEFAULTCARET. DEFAULTCARET can be set to change the initial caret for new processes.

The hotspot of *NEWCARET* indicates which point in the new caret bitmap should be located at the current output position. The previous caret is returned. Note: the bitmap for the caret is not limited to the dimensions CURSORWIDTH by CURSORHEIGHT.

(**CARETRATE** *ONRATE OFFRATE*)   [Function]

Sets the rate at which the caret for the current process will flash. The caret will be visible for *ONRATE* milliseconds, then not visible for *OFFRATE* milliseconds. If *OFFRATE* is NIL then it is set to be the same as *ONRATE*. If *ONRATE* is T, both the "on" and "off" times are set to the value of the variable DEFAULTCARETRATE (initially 333). The previous value of CARETRATE is returned. If the caret is off, CARETRATE return NIL.

Miscellaneous Window Functions

(**CLEARW** *WINDOW*)   [Function]

Fills *WINDOW* with its background texture, changes its coordinate system so that the origin is the lower left corner of the window, sets its X position to the left margin and sets its Y position to the base line of the uppermost line of text, ie. the top of the window less the font ascent.

(**INVERTW** *WINDOW SHADE*)   [Function]

Fills the window *WINDOW* with the texture *SHADE* in INVERT mode. If *SHADE* is NIL, BLACKSHADE is used. INVERTW returns *WINDOW* so that it can be used inside RESETFORM.

(**FLASHWINDOW** *WIN? N FLASHINTERVAL SHADE*) [Function]

Flashes the window *WIN?* by "inverting" it twice. N is the number of times to flash the window (default is 1). *FLASHINTERVAL* is the length of time in milliseconds to wait between flashes (default is 200). *SHADE* is the shade that will be used to invert the window (default is BLACKSHADE).

If *WIN?* is NIL, the whole screen is flashed. In this case, the *SHADE* argument is ignored (can only invert the screen).

(**WHICHW** *X Y*) [Function]

Returns the window which contains the position in screen coordinates of *X* if *X* is a POSITION , the position (*X,Y*) if *X* and *Y* are numbers, or the position of the cursor if *X* is NIL. Returns NIL if the coordinates are not in any window. If they are in more than one window, it returns the uppermost.

Example: (WHICHW) returns the window that the cursor is in.

(**DECODE/WINDOW/OR/DISPLAYSTREAM** *DSORW WINDOWVAR TITLE BORDER*) [Function]

Returns a display stream as determined by the *DSORW* and *WINDOWVAR* arguments. If *DSORW* is a display stream, it is returned. If *DSORW* is a window, its display stream is returned. If *DSORW* is NIL, the litatom *WINDOWVAR* is evaluated. If its value is a window, its display stream is returned. If its value is not a window, *WINDOWVAR* is set to a newly created window (prompting user for region) whose display stream is then returned. If *DSORW* is NEW, the display stream of a newly created window is returned. If a *window* is involved in the decoding, it is opened and if *TITLE* or *BORDER* are given, the *TITLE* or *BORDER* property of the window are reset. The *DSORW* = NIL case is most useful for programs that want to display their output in a window, but want to reuse the same window each time they are called. The non-NIL cases are good for decoding a display stream argument passed to a function.

(**WIDTHIFWINDOW** *INTERIORWIDTH BORDER*) [Function]

Returns the width of the window necessary to have INTERIORWIDTH points in its interior if the width of the border is *BORDER*. If *BORDER* is NIL, the default border size WBorder is used.

(**HEIGHTIFWINDOW** *INTERIORHEIGHT TITLEFLG BORDER*) [Function]

Returns the height of the window necessary to have *INTERIORHEIGHT* points in its interior with a border of *BORDER* and, if *TITLEFLG* is non-NIL, a title. If *BORDER* is NIL, the default border size WBorder is used.

WIDTHIFWINDOW and HEIGHTIFWINDOW are useful for calculating the width and height for a call to GETBOXPOSITION for the purpose of positioning a prospective window.

(**MINIMUMWINDOWSIZE** *WINDOW*)   [Function]

> Returns a dotted pair, the CAR of which is the minimum width *WINDOW* needs and the CDR or which is the minimum height *WINDOW* needs.

> The minimum size is determined by the value of the window property MINSIZE of *WINDOW*. If the value of the MINSIZE window property is NIL, the width is 26 and the height is the height *WINDOW* needs to have its title, border and one line of text visible. If MINSIZE is a dotted pair, it is returned. If it is a litatom, it should be a function which is called with *WINDOW* as its first argument, which should return a dotted pair.

Miscellaneous Window Properties

TITLE   [Window Property]

> Accesses the title of the window. If a title is added to a window whose title is NIL or the title is removed (set to NIL) from a window with a title, the window's exterior (its region on the screen) is enlarged or reduced to accomodate the change without changing the window's interior. For example, (WINDOWPROP WINDOW 'TITLE "Results") changes the title of WINDOW to be "Results". (WINDOWPROP WINDOW 'TITLE NIL) removes the title of WINDOW.

BORDER   [Window Property]

> Accesses the width of the border of the window. The border will have at most 2 point of white (but never more than half) and the rest black. The default border is the value of the global variable WBorder (initially 4).

WINDOWTITLESHADE   [Window Property]

> Accesses the window title shade of the window. If non-NIL, it should be a texture which is used as the "backgound texture" for the title bar on the top of the window. If it is NIL, the value of the global variable WINDOWTITLESHADE (initially BLACKSHADE) is used. Note that black is always used as the background of the title printed in the title bar, so that the letters can be read. The remaining space is painted with the "title shade".

HARDCOPYFN   [Window Property]

> If non-NIL, it should be a function that is called by the window menu command Hardcopy to print the contents of a window. The HARDCOPYFN property is called with two arguments, the window and an image stream to print to. If the window does not have a HARDCOPYFN, the bitmap image of the window (including the border and title) are printed on the file or printer.

DSP   [Window Property]

Value is the display stream of the window. All system functions will operate on either the window or its display stream. This window property cannot be changed using `WINDOWPROP`.

`HEIGHT`   [Window Property]
`WIDTH`   [Window Property]

Value is the height and width of the interior of the window (the usable space not counting the border and title). These window properties cannot be changed using `WINDOWPROP`.

`REGION`   [Window Property]

Value is a region (in screen coordinates) indicating where the window (counting the border and title) is located on the screen. This window property cannot be changed using `WINDOWPROP`.

Example: A Scrollable Window

The following is a simple example showing how one might create a scrollable window.

`CREATE.PPWINDOW` creates a window that displays the pretty printed expression `EXPR`. The window properties `PPEXPR`, `PPORIGX`, and `PPORIGY` are used for saving this expression, and the initial window position. Using this information, `REPAINT.PPWINDOW` simply reinitializes the window position, and prettyprints the expression again. Note that the whole expression is reformatted every time, even if only a small part actually lies within the window. If this window was going to be used to display very large structures, it would be desirable to implement a more sophisticated `REPAINTFN` that only redisplays that part of the expression within the window. However, this scheme would be satisfactory if most of the items to be displayed are small.

`RESHAPE.PPWINDOW` resets the window (and stores the initial window position), calls `REPAINT.PPWINDOW` to display the window's expression, and then sets the `EXTENT` property of the window so that `SCROLLBYREPAINTFN` will be able to handle scrolling and "thumbing" correctly.

```
              (DEFINEQ

              (CREATE.PPWINDOW
                [LAMBDA (EXPR)              (* rrb " 4-OCT-82 12:06")
                                     (* creates a window that displays
                                       a pretty printed expression.)

              (PROG (WINDOW)
                                     (* ask the user for a piece of the
                                       screen and make it into a window.)
                 (SETQ WINDOW (CREATEW NIL "PP window"))
                                     (* put the expression on the
                                       property list of the window so that
                                       the repaint and reshape functions
                                       can access it.)
                 (WINDOWPROP WINDOW (QUOTE PPEXPR) EXPR)
                                     (* set the repaint and reshape
                                       functions.)
```

```
                              (WINDOWPROP WINDOW (QUOTE REPAINTFN)
                                  (FUNCTION REPAINT.PPWINDOW))
                              (WINDOWPROP WINDOW (QUOTE RESHAPEFN)
                                  (FUNCTION RESHAPE.PPWINDOW))
```
(make the scroll function
SCROLLBYREPAINTFN, a system
function that uses the repaint
function to do scrolling.)*
```
                              (WINDOWPROP WINDOW (QUOTE SCROLLFN)
                                  (FUNCTION SCROLLBYREPAINTFN))
```
(call the reshape function to
initially print the expression and
calculate its extent.)*
```
                            (RESHAPE.PPWINDOW WINDOW)
                            (RETURN WINDOW])
```

```
                        (REPAINT.PPWINDOW
                         [LAMBDA (WINDOW REGION)        (* rrb " 4-OCT-82 11:52")
```

(the repainting function for a window with a
pretty printed expression. This repainting
function ignores the region to be repainted
and repaints the entire window.)*

(set the window position to the
beginning of the pretty printing
of the expression.)*
```
                          (MOVETO (WINDOWPROP WINDOW (QUOTE PPORIGX))
                                  (WINDOWPROP WINDOW (QUOTE PPORIGY))
                                  WINDOW)
                          (PRINTDEF (WINDOWPROP WINDOW (QUOTE PPEXPR))
                                  0 NIL NIL NIL WINDOW])
```

```
                        (RESHAPE.PPWINDOW
                          [LAMBDA (WINDOW)              (* rrb " 4-OCT-82 12:01")
```
(the reshape function for a
window with a pretty printed
expression.)*
```
                            (PROG (BTM)
```

(set the position of the window so that the
first character appears in the upper left corner
and save the X and Y for the repaint function.)*
```
                            (DSPRESET WINDOW)
                            (WINDOWPROP WINDOW (QUOTE PPORIGX)
                                  (DSPXPOSITION NIL WINDOW))
                            (WINDOWPROP WINDOW (QUOTE PPORIGY)
                                  (DSPYPOSITION NIL WINDOW))
```
(call the repaint function to
pretty print the expression in*

the newly cleared window.)
```
(REPAINT.PPWINDOW WINDOW)
```

(* save the region actually covered by the pretty
printed expression so that the scrolling routines
will know where to stop. The pretty printing of
the expression does a carriage return after the
last piece of the expression printed so that the
current position is the base line of the next line
of text. Hence the last visible piece of the
expression (BTM) is the ending position plus the
height of the font above the base line (its ASCENT).)

```
(WINDOWPROP WINDOW (QUOTE EXTENT)
   create REGION
          LEFT ← 0
          BOTTOM ← [SETQ BTM (IPLUS
                  (DSPYPOSITION NIL WINDOW)
                  (FONTPROP WINDOW (QUOTE ASCENT]
          WIDTH ←(WINDOWPROP WINDOW (QUOTE WIDTH))
          HEIGHT ←(IDIFFERENCE
                  (WINDOWPROP WINDOW (QUOTE HEIGHT))
                  BTM])
)
```

Menus

A menu is basically a means of selecting from a list of items. The system provides common layout and interactive user selection mechanisms, then calls a user-supplied function when a selection has been confirmed. The two major constituents of a menu are a list of items and a "when selected function." The label that appears for each item is the item itself for non-lists, or its CAR if the item is a list. In addition, there are a multitude of different formatting parameters for specifying font, size, and layout. When a menu is created, its unspecified fields are filled with defaults and its screen image is computed and saved.

Menus can be either pop up or fixed. If fixed menus are used, the menu must be included in a window.

(**MENU** *MENU POSITION RELEASECONTROLFLG* −)   [Function]

This function provides menus that pop up when they are used. It displays *MENU* at *POSITION* (in screen coordinates) and waits for the user to select an item with a mouse key. Before any mouse key is pressed, the item the mouse is over is boxed. After any key is down, the selected menu item is video reversed. When all keys are released, *MENU*'s WHENSELECTEDFN field is called with four arguments: (1) the item selected, (2) the menu, (3) the last mouse key released (LEFT, MIDDLE, or RIGHT), and (4) the reverse list of superitems rolled through when selecting the item and *MENU* returns its

value. If no item is selected, *MENU* returns NIL. If *POSITION* is NIL, the menu is brought up at the value from *MENU*'s MENUPOSITION field, if it is a *POSITION*, or at the current cursor position. The orientation of *MENU* with respect to the specified position is determined by its MENUOFFSET field.

If *RELEASECONTROLFLG* is NIL, this process will retain control of the mouse. In this case, if the user lets the mouse key up outside of the menu, *MENU* return NIL. (Note: this is the standard way of allowing the user to indicate that they do not want to make the offered choice.) If *RELEASECONTROLFLG* is non-NIL, this process will give up control of the mouse when it is outside of the menu so that other processes can be run. In this case, clicking outside the menu has no effect on the call to *MENU*. If the menu is closed (for example, by right buttoning in it and selecting "Close" from the window menu), *MENU* returns NIL. Programmers are encouraged to provide a menu item such as "cancel" or "abort" which gives users a positive way of indicating "no choice".

> Note: A "released" menu will stay visible (on top of the window stack) until it is closed or an item is selected.

(**ADDMENU** *MENU WINDOW POSITION DONTOPENFLG*)   [Function]

This function provides menus that remain active in windows. ADDMENU displays *MENU* at *POSITION* (in window coordinates) in *WINDOW*. If the window is too small to display the entire menu, the window is made scrollable. When an item is selected, the value of the WHENSELECTEDFN field of *MENU* is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse button that the item was selected with (LEFT, MIDDLE, or RIGHT). More than one menu can be put in a window, but a menu can only be added to one window at a time. ADDMENU returns the window into which *MENU* is placed.

If *WINDOW* is NIL, a window is created at the position specified by *POSITION* (in screen coordinates) that is the size of *MENU*. If a window is created, it will be opened unless *DONTOPENFLG* is non-NIL. If *POSITION* is NIL, the menu is brought up at the value of *MENU*'s MENUPOSITION field (in window coordinates), if it is a position, or else in the lower left corner of *WINDOW*. If both *WINDOW* and *POSITION* are NIL, a window is created at the current cursor position.

Warning: ADDMENU resets several of the window properties of *WINDOW*.   The CURSORINFN, CURSORMOVEDFN, and BUTTONEVENTFN window properties are replaced with MENUBUTTONFN, so that *MENU* will be active. MENUREPAINTFN is added to the REPAINTFN window property to update the menu image if the window is redisplayed. The SCROLLFN window property is changed to SCROLLBYREPAINTFN if the window is too small for the menu, to make the window scroll.

(**DELETEMENU** *MENU CLOSEFLG FROMWINDOW*)   [Function]

This function removes *MENU* from the window *FROMWINDOW*. If *MENU* is the only menu in the window and *CLOSEFLG* is non-NIL, its window will be closed (by CLOSEW).

If *FROMWINDOW* is NIL, the list of currently open windows is searched for one that contains *MENU*. If none is found, DELETEMENU does nothing.

Menu Fields

A menu is a datatype with the following fields:

ITEMS   [Menu Field]

The list of items to appear in the menu. If an item is a list, its CAR will appear in the menu. If the item (or its CAR) is a bitmap, the bitmap will be displayed in the menu. The default selection functions interpret each item as a list of three elements: a label, a form whose value is returned upon selection, and a help string that is printed in the prompt window when the user presses a mouse key with the cursor pointing to this item. The default subitem function interprets the fourth element of the list. If it is a list whose CAR is the litatom SUBITEMS , the CDR is taken as a list of subitems.

SUBITEMFN   [Menu Field]

A function to be called to determine if an item has any subitems. If an item has subitems and the user rolls the cursor out the right of that item, a submenu with that item's subitems in it pops up. If the user selects one of the items from the submenu, the selected subitem is handled as if it were selected from the main menu. If the user rolls out of the submenu to the left, the submenu is taken down and selection resumes from the main menu.

An item with subitems is marked in the menu by a grey, right pointing triangle following the label.

The function is called with two arguments: (1) the menu and (2) the item. It should return a list of the subitems of this item if any. (It is called twice to compute the menu image and each time the user rolls out of the item box so it should be moderately efficient. The default SUBITEMFN, DEFAULTSUBITEMFN, checks to see if the item is a list whose fourth element is a list whose CAR is the litatom SUBITEMS and if so, returns the CDR of it.

For example:

```
(create MENU
    ITEMS ← '(AAAA (BBBB 'BBBB "help string for
BBBB"
            (SUBITEMS BBBB1 BBBB2 BBBB3))))
```

will create a menu with items A and B in which B will have subitems B1, B2 and B3. The following picture below shows this menu as it first appears:
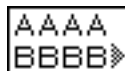
The following picture shows the submenu, with the item BBBB3 selected by the cursor (🖚):
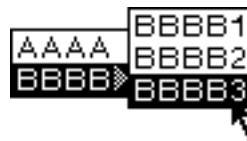
WHENSELECTEDFN   [Menu Field]

A function to be called when an item is selected. The function is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (LEFT, MIDDLE, or RIGHT). The default function DEFAULTWHENSELECTEDFN evaluates and returns the value of the second element of the item if the item is a list of at least length 2. If the item is not a list of at least length 2, DEFAULTWHENSELECTEDFN returns the item.

Note: If the menu is added to a window with ADDMENU, the default WHENSELECTEDFN is BACKGROUNDWHENSELECTEDFN, which is the same as DEFAULTWHENSELECTEDFN except that EVAL.AS.PROCESS is used to evaluate the second element of the item, instead of tying up the mouse process.

WHENHELDFN   [Menu Field]

The function which is called when the user has held a mouse key on an item for MENUHELDWAIT milliseconds (initially 1200). The function is called with three arguments: (1) the item selected, (2) the menu, and (3) the mouse key that the item was selected with (LEFT, MIDDLE, or RIGHT). WHENHELDFN is intended for prompting users. The default is DEFAULTMENUHELDFN which prints (in the prompt window) the third element of the item or, if there is not a third element, the string "This item will be selected when the button is released."

WHENUNHELDFN   [Menu Field]

If WHENHELDFN was called, WHENUNHELDFN will be called: (1) when the cursor leaves the item, (2) when a mouse key is released, or (3) when another key is pressed. The function is called with the same three argument values used to call WHENHELDFN. The default WHENUNHELDFN is the function CLRPROMPT, which just clears the prompt window.

MENUPOSITION   [Menu Field]

The position of the menu to be used if the call to MENU or ADDMENU does not specify a position. For popup menus, this is in screen coordinates. For fixed menus, it is in the coordinates of the window the menu is in. The point within the menu image that is placed at this position is determined by MENUOFFSET. If MENUPOSITION is NIL, the menu will be brought up at the cursor position.

MENUOFFSET   [Menu Field]

The position in the menu image that is to be located at `MENUPOSITION`. The default offset is (0,0). For example, to bring up a menu with the cursor over a particular menu item, set its `MENUOFFSET` to a position within that item and set its `MENUPOSITION` to `NIL`.

MENUFONT   [Menu Field]

The font in which the items will be appear in the menu. Default is the value of `MENUFONT`.

TITLE   [Menu Field]

If non-`NIL`, the value of this field will appear as a title in a line above the menu.

MENUTITLEFONT   [Menu Field]

The font in which the title of the menu will be appear. If this is `NIL`, the title will be in the same font as window titles. If it is `T`, it will be in the same font as the menu items.

CENTERFLG   [Menu Field]

If non-`NIL`, the menu items are centered; otherwise they are left-justified.

MENUROWS   [Menu Field]
MENUCOLUMNS   [Menu Field]

These fields control the shape of the menu in terms of rows and columns. If `MENUROWS` is given, the menu will have that number of rows. If `MENUCOLUMNS` is given, the menu will have that number of columns. If only one is given, the other one will be calculated to generate the minimal rectangular menu. (Normally only one of `MENUROWS` or `MENUCOLUMNS` is given.) If neither is given, the items will be in one column.

ITEMHEIGHT   [Menu Field]

The height of each item box in the menu. If not specified, it will be the maximum of the height of the `MENUFONT` and the heights of any bitmaps appearing as labels.

ITEMWIDTH   [Menu Field]

The width of each item box in the menu. If not specified, it will be the width of the largest item in the menu.

MENUBORDERSIZE   [Menu Field]

The size of the border around each item box. If not specified, 0 (no border) is used.

MENUOUTLINESIZE   [Menu Field]

The size of the outline around the entire menu. If not specified, a maximum of 1 and the `MENUBORDERSIZE` is used.

CHANGEOFFSETFLG   [Menu Field]

(popup menus only) If CHANGEOFFSETFLG is non-NIL, the position of the menu offset is set each time a selection is confirmed so that the menu will come up next time in the same position relative to the cursor. This will cause the menu to reappear in the same place on the screen if the cursor has not moved since the last selection. This is implemented by changing the MENUOFFSET field on each use. If CHANGEOFFSETFLG is the atom X or the atom Y, only the X or the Y coordinate of the MENUOFFSET field will be changed. For example, by setting the MENUOFFSET position to (-1,0) and setting CHANGEOFFSETFLG to Y, the menu will pop up so that the cursor is just to the left of the last item selected. This is the setting of the window command menus.

The following fields are read only.

IMAGEHEIGHT   [Menu Field]

Returns the height of the entire menu.

IMAGEWIDTH   [Menu Field]

Returns the width of the entire menu.

Miscellaneous Menu Functions

(**MAXMENUITEMWIDTH** *MENU*)   [Function]

Returns the width of the largest menu item label in the menu *MENU*.

(**MAXMENUITEMHEIGHT** *MENU*)   [Function]

Returns the height of the largest menu item label in the menu *MENU*.

(**MENUREGION** *MENU*)   [Function]

Returns the region covered by the image of *MENU* in its window.

(**WFROMMENU** *MENU*)   [Function]

Returns the window *MENU* is located in, if it is in one; NIL otherwise.

(**DOSELECTEDITEM** *MENU ITEM BUTTON*)   [Function]

Calls *MENU*'s WHENSELECTEDFN on *ITEM* and *BUTTON*. It provides a programmatic way of making a selection. It does not change the display.

(**MENUITEMREGION** *ITEM MENU*)   [Function]

Returns the region occupied by *ITEM* in *MENU*.

(**SHADEITEM** *ITEM MENU SHADE DS/W*)   [Function]

Shades the region occupied by *ITEM* in *MENU*. If *DS/W* is a display stream or a window, it is assumed to be where *MENU* is displayed. Otherwise, WFROMMENU is called to locate the

window *MENU* is in. Shading is persistent, and is reapplied when the window the menu is in gets redisplayed. To unshade an item, call with a *SHADE* of 0.

(**PUTMENUPROP** *MENU PROPERTY VALUE*) [Function]

Stores the property *PROPERTY* with the value *VALUE* on a property list in the menu *MENU*. The user can use this property list for associating arbitrary data with a menu object.

(**GETMENUPROP** *MENU PROPERTY*) [Function]

Returns the value of the *PROPERTY* property of the menu *MENU*.

Examples of Menu Use

Example: A simple menu:

```
(MENU (create MENU ITEMS _ '((YES T) (NO (QUOTE
NIL))) ))
```

Creates a menu with items `YES` and `NO` in a single vertical column:

```
YES
NO
```

If `YES` is selected, `T` will be returned. Otherwise, `NIL` will be returned.

Example: A simple menu, with centering:

```
(MENU (create MENU TITLE ← "Foo?"
        ITEMS ← '((YES T "Adds the Foo feature.")
                  (NO 'NO "Removes the Foo feature."))
              CENTERFLG ← T))
```

Creates a menu with a title `Foo?` and items `YES` and `NO` centered in a single vertical column:

```
Foo?
YES
NO
```

The strings following the `YES` and `NO` are help strings and will be printed if the cursor remains over one of the items for a period of time. This menu differs from the one above in that it distinquishes the `NO` case from the case where the user clicked outside of the menu. If the user clicks outside of the menu, `NIL` is returned.

Example: A multi-column menu:

```
(create MENU ITEMS ← '(1 2 3 4 5 6 7 8 9 * 0 #)
             CENTERFLG ← T
```

```
MENUCOLUMNS ← 3
MENUFONT ← (FONTCREATE 'MODERN 10 'BOLD)
ITEMHEIGHT ← 15
ITEMWIDTH ← 15
CHANGEOFFSETFLG ← T)
```

Creates a touch-tone-phone number pad with the items in 15 by 15 boxes printed in Modern 10 bold font:

If used in pop up mode, its first use will have the cursor in the middle. Subsequent use will have the cursor in the same relative location as the previous selection.

Example: A program using a previously-saved menu:

```
(SELECTQ [MENU
    (COND ((type? MENU FOOMENU)
             (* use previously computed menu.)
           FOOMENU)
          (T (* create and save the menu)
         (SETQ FOOMENU
           (create MENU
                ITEMS ← '((A 'A-SELECTED "prompt string
    for A")
                          (B 'B-SELECTED "prompt string for B"]
      (A-SELECTED (* if A is selected) (DOATHING))
      (B-SELECTED (* if B is selected) (DOBTHING))
       (PROGN (* user selected outside the menu) NIL)))
```

This expression displays a pop up menu with two items, A and B, and waits for the user to select one. If A is selected, DOATHING is called. If B is selected, DOBTHING is called. If neither of these is selected, the form returns NIL.

The purpose of this example is to show some good practices to follow when using menus. First, the menu is only created once, and saved in the variable FOOMENU. This is more efficient if the menu is used more than once. Second, all of the information about the menu is kept in one place, which makes it easy to understand and edit. Third, the forms evaluated as a result of selecting something from the menu are part of the code and hence will be known to masterscope (as opposed to the situation if the forms were stored as part of the items). Fourth, the items in the menu have help strings for the user. Finally, the code is commented (always worth the trouble).

Free Menus

Free Menus are powerful and flexible menus that are useful for applications needing menus with different types of items, including command items, state items, and items that can be edited. A Free Menu is part of a window. It can can be opened and closed as desired, or attached as a control menu to the application window.

Making a Free Menu

A Free Menu is built from a description of the contents and layout of the menu. As a Free Menu is simply a group of items, a Free Menu Description is simply a specification of a group of items. Each group has properties associated with it, as does each Free Menu Item. These properties specify the format of the items in the group, and the behavior of each item. The function FREEMENU takes a Free Menu Description, and returns a closed window with the Free Menu in it.

The easiest way to make a Free Menu is to define a specific function which calls FREEMENU with the Free Menu Description in the function. This function can then also set up the Free Menu window as required by the application. The Free Menu Description is saved as part of the specific function when the application is saved. Alternately, the Free Menu Description can be saved as a variable in your file; then just call FREEMENU with the name of the variable. This may be a more difficult alternative if the backquote facility is used to build the Free Menu Description.

Free Menu Formatting

A Free Menu can be formatted in one of four ways. The items in any group can be automatically laid out in rows, in columns, or in a table, or else the application can specify the exact location of each item in the group. Free Menu keeps track of the region that a group of items occupies, and items can be justified within that region. This way an item can be automatically positioned at one of the nine justification locations, top-left, top-center, top-right, middle-left, etc.

Free Menu Description

A Free Menu Description, specifying a group of items, is a list structure. The first entry in the list is an optional list of the properties for this group of items. This entry is in the form:

```
(PROPS    <PROP> <VALUE> <PROP> <VALUE> ...)
```

The keyword PROPS determines whether or not the optional group properties list is specified..

One important group property is FORMAT. The four types of formatting, ROW, TABLE, COLUMN, or EXPLICIT, determine the syntax of the rest of the Free Menu Description. When using EXPLICIT formatting, the rest of the description is any number of Item Descriptions which have LEFT and BOTTOM properties specifying the position of the item in the menu. The syntax is:

```
((PROPS FORMAT EXPLICIT ...)
 <ITEM DESCRIPTION>
 <ITEM DESCRIPTION> ...)
```

When using `ROW` or `TABLE` formatting, the rest of the description is any number of item groups, each group corresponding to a row in the menu. These groups are identical in syntax to an `EXPLICIT` group description. The groups have an optional `PROPS` list and any number of Item Descriptions. The items need not have `LEFT` and `BOTTOM` properties, as the location of each item is determined by the formatter. However, the order of the rows and items is important. The menu is laid out top to bottom by row, and left to right within each row. The syntax is:

```
((PROPS FORMAT ROW ...)      ; props of this group
 (<ITEM DESCRIPTION>         ; items in first row
  <ITEM DESCRIPTION> ...)
  ((PROPS ...)               ; props of second row
   <ITEM DESCRIPTION>        ; items in second row
   <ITEM DESCRIPTION> ...))
```

(The comments above only describe the syntax.)

For `COLUMN` formatting, the syntax is identical to that of `ROW` formatting. However, each group of items corresponds to a column in the menu, rather than a row. The menu is laid out left to right by column, top to bottom within each column.

Finally, a Free Menu Description can have recursively nested groups. Anywhere the description can take an Item Description, it can take a group, marked by the keyword `GROUP`. A nested group inherits all of the properties of its mother group, by default. However, any of these properties can be overridden in the nested groups `PROPS` list, including the `FORMAT`. The syntax is:

```
(                 ; no PROPS list, default row format
(<ITEM DESCRIPTION>          ; first in row
(GROUP          ; nested group, second in row
   (PROPS FORMAT COLUMN ...) ; optional props
   (<ITEM DESCRIPTION> ...)  ; first column
   (<ITEM DESCRIPTION> ...))
    <ITEM DESCRIPTION>))     ; third in row
```

Here is an example of a simple Free Menu Description for a menu which might provide access to a simple data base:

```
(((LABEL LOOKUP SELECTEDFN MYLOOKUPFN)
  (LABEL EXIT SELECTEDFN MYEXITFN))
 ((LABEL Name: TYPE DISPLAY) (LABEL "" TYPE EDIT ID NAME))
 ((LABEL Address: TYPE DISPLAY) (LABEL "" TYPE EDIT ID ADDRESS))
 ((LABEL Phone: TYPE DISPLAY)
```

```
   (LABEL "" TYPE EDIT LIMITCHARS MYPHONEP ID PHONE)))
```

This menu has two command buttons, LOOKUP and EXIT, and three edit fields, with IDs NAME, PHONE, and ADDRESS. The Edit items are initialized to the empty string, as in this example they need no other initial value. The user could select the Name: prompt, type a person's name, and then press the LOOKUP button.   The function MYLOOKUPFN would be called.   That function would look at the NAME Edit item, look up that name in the data base, and   fill in the rest of the fields appropriately. The PHONE item has MYPHONEP as a LIMITCHARS function. This function would be called when   editing the phone number, in order to restrict input to a valid phone number. After looking up Perry, the Free Menu might look like:

```
┌──────────────────────────────┐
│ LOOKUP  EXIT                  │
│ Name: Herbert Q Perry         │
│ Address:  13 Middleperry Dr   │
│ Phone:  (411) 767-1234        │
└──────────────────────────────┘
```

Here is a more complicated example:

```
((PROPS FONT (MODERN 10))
 ((LABEL Example FONT (MODERN 10 BOLD) HJUSTIFY CENTER))
 ((LABEL NORTH) (LABEL SOUTH) (LABEL EAST) (LABEL WEST))
 ((PROPS ID ROW3 BOX 1)
  (LABEL ONE) (LABEL TWO) (LABEL THREE))
 ((PROPS ID ROW4)
  (LABEL ONE ID ALPHA)
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
     ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT
T))
     (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
     (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
       ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
           INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
     (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
  (LABEL THREE)))
```

which will produce the following Free Menu:
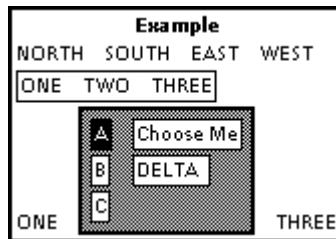
And if the Free Menu were formatted as a Table, instead of in Rows, it would look like:
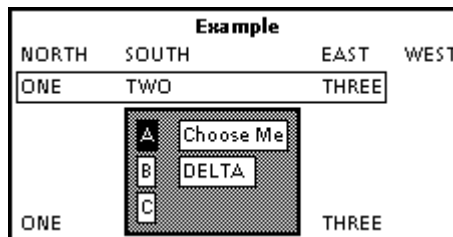
The following breakdown of the example explains how each part contributes to the Free Menu shown above.

```
(PROPS FONT (MODERN 10))
```

This line specifies the properties of the group that is the entire Free Menu. These properties are described in Section 28.7.4, Free Menu Group Properties. In this example, all items in the Free Menu, unless otherwise specified, will be in Modern 10.

```
((LABEL Example FONT (MODERN 10 BOLD) HJUSTIFY CENTER))
```

This line of the Free Menu Description describes the first row of the menu. Since the `FORMAT` specification of a Free Menu is, by default, `ROW` formatting, this line sets the first row in the menu. If the menu were in `COLUMN` formatting, this position in the description would specify the first column in the menu.

In this example the first row contains only one item. The item is, by default, a type `MOMENTARY` item. It has its own Font declaration `(FONT (MODERN 10 BOLD))`, that overrides the font specified for the Free Menu as a whole, so the item appears bolded.

Finally, the item is justified, in this case centered. The `HJUSTIFY` Item Property indicates that the item is to be centered horizontally within its row.

```
((LABEL NORTH) (LABEL SOUTH) (LABEL EAST) (LABEL WEST))
```

This line specifies the second row of the menu. The second row has four very simple items, labeled NORTH, SOUTH, EAST, and WEST next to each other within the same row.

```
((PROPS ID ROW3 BOX 1)
 (LABEL ONE) (LABEL TWO) (LABEL THREE))
```

The third row in the menu is similar to the second row, except that it has a box drawn around it. The box is specified in the PROPS declaration for this row. Rows (and columns) are just like Groups in that the first thing in the declaration can be a list of properties for that row. In this case the row is named by giving it an ID property of ROW3. It is useful to name your groups if you want to be able to access and modify their properties later (via the function FM.GROUPPROP). It is boxed by specifying the BOX property with a value of 1, meaning draw the box one dot wide.

```
 ((PROPS ID ROW4)
  (LABEL ONE ID ALPHA)
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
   ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
        (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
        (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
       ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
         INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
        (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
  (LABEL THREE)))
```

This part of the description specifies the fourth row in the menu. This row consists of: an item labelled ONE, a group of items, and an item labelled THREE. That is, Free Menu thinks of the group as an entry, and formats the rest of the row just as it it were a large item.

```
 (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
  ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
   (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
   (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
  ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
    INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
   (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
```

The second part of this row is a nested group of items. It is declared as a group by placing the keyword GROUP as the first word in the declaration. A group can be declared anywhere a Free Menu Description can take a Free Menu Item Description (as opposed to a row or column declaration).

The first thing in what would have been the second item declaration in this row is the keyword GROUP. Following this keyword comes a normal group description, starting with an optional list of properties, and followed by any number of things to go in the group (based on the format of the group).

This group's Props declaration is:

```
(PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4).
```

It specifies that the group is to be formatted as a number of columns (instead of rows, the default). The entire group will have a background shade of 23130, and a box of width 2 around it, as you can see in the sample menu. The BOXSPACE declaration tells Free Menu to leave an extra four dots of room between the edge of the group (ie the box around the group) and the items in the group.

The first column of this group is a Collection of NWAY items:

```
((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
 (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
 (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
```

The three items, labelled A, B, and C are all declared as NWAY items, and are also specified to belong to the same NWAY Collection, Col1. This is how a number of NWAY items are collected together. The property NWAYPROPS (DESELECT T) on the first NWAY item specifies that the Col1 Collection is to have the Deselect property enabled. This simply means that the NWAY collection can be put in the state where none of the items (A, B, or C) are selected (highlighted). Additionally, each item is declared with a box whose width is one dot (pixel) around it.

The second column in this nested group is specified by:

```
((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
     INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
 (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35))
```

Column two contains two items, a STATE item and a DISPLAY item. The STATE item is labelled "Choose Me." A Label can be a string or a bitmap, as well as an atom. Selecting the STATE item will cause a pop-up menu to appear with two choices for the state of the item, BRAVO and DELTA. The items to go in the pop-up menu are designated by the MENUITEMS property.

The pop-up menu would look like:
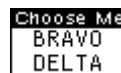
The initial state of the "Choose Me" item is designated to be DELTA by the INITSTATE Item Property. The initial state can be anything; it does not have to be one of the items in the pop-up menu.

Next, the STATE item is Linked to a DISPLAY item, so that the current state of the item will be displayed in the Free Menu. The link's name is DISPLAY (a special link name for

STATE items), and the item linked to is described by the Link Description, (GROUP ALPHA). Normally the linked item can just be described by its ID. But in this case, there is more than one item whose ID is ALPHA (for the sake of this example), specifically the first item in the fourth row and the display item in this nested group. The form (GROUP ALPHA) tells Free Menu to search for an item whose ID is ALPHA, limiting the search to the items that are within this lexical group. The lexical group is the smallest group that is declared with the GROUP keyword (i.e., not row and column groups) that contains this item declaration. So in this case, Free Menu will link the STATE item to the DISPLAY item, rather than the first item in the fourth row, since *that* item is outside of the nested group. For further discussion of linking items, see Section 28.7.12, Free Menu Item Links.

Now, establish the DISPLAY item:

(TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)

We have given it the ID of Alpha that the above STATE item uses in finding the proper DISPLAY item to link to. This display item is used to display the current state of the item "Choose Me." Every item is required to have a Label property specified, but the label for this DISPLAY item will depend on the state of "Choose Me." That is, when the state of the "Choose Me" item is changed from DELTA to BRAVO, the label of the DISPLAY item will also change. The null string serves to hold the place for the changeable label.

A box is specified for this item. Since the label is the empty string, Free Menu would draw a very small box. Instead, the MAXWIDTH property indicates that the label, whatever it becomes, will be limited to a stringwidth of 35. The width restriction of 35 was chosen because it is big enough for each of the possible labels for this display item. So Free Menu draws the box big enough to enclose any item within this width restriction.

Finally we specify the final item in row four:

(LABEL THREE)

Free Menu Group Properties

Each group has properties. Most group properties are relevant and should be set in the group's PROPS list in the Free Menu Description. User properties can be freely included in the PROPS list. A few other properties are set up by the formatter. The macros FM.GROUPPROP or FM.MENUPROP allow access to group properties after the Free Menu is created.

ID The identifier of this group. Setting the group ID is desirable, for example, if the application needs to get handles on items in particular groups, or access group properties.

FORMAT One of ROW, COLUMN, TABLE, or EXPLICIT. The default is ROW.

FONT A font description of the form (FAMILY SIZE FACE), or a FONTDESCRIPTOR data type. This will be the default font for each item

in this group. The default font of the top group is the value of the variable `DEFAULTFONT`.

COORDINATES — One of `GROUP` or `MENU`. This property applies only to `EXPLICIT` formatting. If `GROUP`, the items in the `EXPLICIT` group are positioned in coordinates relative to the lower left corner of the group, as determined by the mother group. If `MENU`, which is the default, the items are positioned relative to the lower left corner of the menu.

LEFT — Specifies a left offset for this group, pushing the group to the right.

BOTTOM — Specifies a bottom offset for this group, pushing the group up.

ROWSPACE — Specifies the number of dots between rows in this group.

COLUMNSPACE — Specifies the number of dots between columns in this group.

BOX — Specifies the number of dots in the box around this group of items.

BOXSHADE — Specifies the shade of the box.

BOXSPACE — Specifies the number of bits between the box and the items.

BACKGROUND — The background shade of this group. Nested groups inherit this background shade, but items in this group and nested groups do not. This is because, in general, it is difficult to read text on a background, so items appear on a white background by default. This can be overridden by the `BACKGROUND` Item Property.

Other Group Properties

The following group properties are set up and maintained by Free Menu. The application should probably not change any of these properties.

ITEMS — A list of the items in the group.

REGION — The region that is the extent of the items in the group.

MOTHER — The `ID` of the group that is the mother of this group.

DAUGHTERS — A list of `ID` of groups which are daughters to this group.

Free Menu Items

Each Free Menu Item is stored as an instance of the data type `FREEMENUITEM`. Free Menu Items can be thought of as objects, each item having its own particular properties, such as its type, label, and mouse event functions. A number of useful item types, described in Section 28.7.11, Predefined Item Types, are predefined by Free Menu. New types of items can be defined by the application, using

Display items as a base. Each Free Menu Item is created from a Free Menu Item Description when the Free Menu is created.

> CAUTION: Edit (and thus Number) Freemenu Items do not perform well when boxed or when there is another item to the right in the same row. The display to the right of the edit item may be corrupted under editing and fm.changelabel operations.

Free Menu Item Descriptions

A Free Menu Item Description is a list in property list format, specifying the properties of the item. For example:

```
(LABEL Refetch SELECTEDFN MY.REFETCHFN)
```

describes a MOMENTARY item labelled Refetch, with the function MY.REFETCHFN to be called when the item is selected. None of the property values in an item description are evaluated. When constructing Free Menu descriptions that incorporate evaluated expressions (for example labels that are bitmaps) it is helpful to use the backquote facility. For instance, if the value of the variable MYBITMAP is a bitmap, then

```
(FREEMENU '(((LABEL A) (LABEL ,MYBITMAP))))
```

would create a Free Menu of one row, with two items in that row, the second of which has the value of MYBITMAP as its label.

Free Menu Item Properties

The following Free Menu Item Properties can be set in the Item Description. Any other properties given in an Item Description will be treated as user properties, and will be saved on the USERDATA property of the item.

> TYPE   The type of the item. Choose from one of the Free Menu Item type keywords MOMENTARY, TOGGLE,   3STATE,   STATE,   NWAY, EDITSTART, EDIT, NUMBER, or DISPLAY. The default is MOMENTARY.
>
> LABEL   An atom, string, or bitmap. Bitmaps are always copied, so that the original will not be changed. This property must be specified for every item.
>
> FONT   The font in which the item appears. The default is the font specified for the group containing this item. Can be a font description of the form (FAMILY SIZE FACE), or a FONTDESCRIPTOR data type.
>
> ID   May be used to specify a unique identifier for this item, but is not necessary.

`LEFT` and `BOTTOM`	When `ROW`, `COLUMN`, or `TABLE` formatting, these specify offsets, pushing the item right and up, respectively, from where the formatter would have put the item. In `EXPLICIT` formatting, these are the actual coordinates of the item, in the coordinate system given by the group's `COORDINATES` property.
`HJUSTIFY`	Indicates horizontal justification type: `LEFT`, `CENTER`, or `RIGHT`. Specifies that this item is to be horizontally justified within the extent of its group. Note that the main group, as opposed to the smaller row or column group, is used.
`VJUSTIFY`	Specifies that this item is to be vertically justified. Values are `TOP`, `MIDDLE`, or `BOTTOM`.
`HIGHLIGHT`	Specifies the highlighted looks of the item, that is, how the item changes when a mouse event occurs on it. See Section 28.7.12, Free Menu Item Highlighting, for more details on highlighting.
`MESSAGE`	Specifies a string that will be printed in the prompt window after a mouse cursor selects this item for `MENUHELDWAIT` milliseconds. Or, if an atom, treated as a function to get the message. The function is passed three arguments, `ITEM`, `WINDOW`, and `BUTTONS`, and should return a string. The default is a message appropriate to the type of the item.
`INITSTATE`	Specifies the initial state of the item. This is only appropriate to `TOGGLE`, `3STATE`, and `STATE` items.
`MAXWIDTH`	Specifies the width allowed for this item. The formatter will leave enough space after the item for the item to grow to this width without collisions.
`MAXHEIGHT`	Similar to `MAXWIDTH`, but in the vertical dimension.
`BOX`	Specifies the number of bits in the box around this item. Boxes are made around `MAXWIDTH` and `MAXHEIGHT` dimensions. If unspecified, no box is drawn.
`BOXSHADE`	Specifies the shade that the box is drawn in. The default is `BLACKSHADE`.
`BOXSPACE`	Specifies the number of bits between the box and the label. The default is one bit.
`BACKGROUND`	Specifies the background shade on which the item appears. The default is `WHITESHADE`, regardless of the group's background.
`LINKS`	Can be used to link this item to other items in the Free Menu.

Mouse Properties

The following properties provide a way for application functions to be called under certain mouse events. These functions are called with the `ITEM`, the `WINDOW`, and the `BUTTONS` passed as arguments. These application functions do not interfere with any Free Menu system functions that take care of handling the different item types. In each case, though, the application function is called

after the system function. The default for all of these functions is `NILL`. The value of each of the following properties can be the name of a function, or a lambda expression.

SELECTEDFN
: Specifies the function to be called when this item is selected. The `Edit` and `EditStart` items cannot have a `SELECTEDFN`. See the Edit Free Menu item description in Section 28.7.11, Predefined Item Types, for more information.

DOWNFN
: Specifies the function to be called when the item is selected with the mouse cursor.

HELDFN
: Specifies the function to be called repeatedly when the item is selected with the mouse cursor.

MOVEDFN
: Specifies the function to be called when the mouse cursor moves off this item (mouse buttons are still depressed).

System Properties

The following Free Menu Item properties are set and maintained by Free Menu. The application should probably not change these properties directly.

GROUPID
: Specifies the `ID` of the smallest group that the item is in. For example, in a row formatted group, the item's `GROUPID` will be set to the `ID` of the row that the item is in, not the `ID` of the whole group.

STATE
: Specifies the current state of `TOGGLE`, `3STATE`, or `STATE` items. The state of an `NWAY` item behaves like that of a toggle item.

BITMAP
: Specifies the bitmap from which the item is displayed.

REGION
: Specifies the region of the item, in window coordinates. This is used for locating the display position, as well as determining the mouse sensitive region of the item.

MAXREGION
: Specifies the maximum region the item may occupy, determined by the `MAXWIDTH` and `MAXHEIGHT` properties (see Section 28.7.8, Free Menu item Properties). This is used by the formatter and the display routines.

SYSDOWNFN

SYSMOVEDFN

SYSSELECTEDFN
: These are the system mouse event functions, set up by Free Menu according to the item type. These functions are called before the mouse event functions, and are used to implement highlighting, state changes, editing, etc.

USERDATA
: Specifies how any other properties are stored on this list in property list format. This list should probably not need to be manipulated directly.

Predefined Item Types

MOMENTARY   [Free Menu Item]

> `MOMENTARY` items are like command buttons. When the button is selected, its associated function is called.

TOGGLE   [Free Menu Item]

> Toggle items are simple two-state buttons. When pressed, the button is highlighted; it stays that way until pressed again. The states of a toggle button are `T` and `NIL`; the initial state is `NIL`.

3STATE   [Free Menu Item]

> `3STATE` items rotate through `NIL`, `T`, and `OFF`, states each time they are pressed. The default looks of the `OFF` state are with a diagonal line through the button, while `T` is highlighted, and `NIL` is normal. The default initial state is `NIL`.

> The following Item Property applies to `3STATE` items:

> > OFF Specifies the looks of a `3STATE` item in its `OFF` state. Similar to `HIGHLIGHT`. The default is that the label gets a diagonal slash through it.

> > > NOTE: If you specify special highlighting (a different bitmap of string) for Toggle or 3State items AND use this item in a group formatted as a Column or a Table, the highlight looks of the item may not appear in the correct place.

STATE   [Free Menu Item]

> `STATE` items are general multiple state items. The following Item Property determines how the item changes state:

> CHANGESTATE This Item Property can be changed at any time to change the effect of the item. If a `MENU` data type, this menu pops up when the item is selected, and the user can select the new state. Otherwise, if this property is given, it is treated as a function name, which is passed three arguments, `ITEM`, `WINDOW`, and `BUTTONS`. This function can do whatever it wants, and is expected to return the new state (an atom, string, or bitmap), or `NIL`, indicating the state should not change. The state of the item can automatically be indicated in the Free Menu, by setting up a `DISPLAY` link to a `DISPLAY` item in the menu (see Section 28.7.13, Free Menu Item Links). If such a link exists, the label of the `DISPLAY` item will be changed to the new state. The possible states are not restricted at all, with the exception of selections from a pop-up menu. The state can be changed to any atom, string, or bitmap, manually via `FM.CHANGESTATE`.

> The following Item Properties are relevant to `STATE` items when building a Free Menu:

> MENUITEMS If specified, should be a list of items to go in a pop-up menu for this item. Free Menu will build the menu and save it as the `CHANGESTATE` property of the item.

MENUFONT   The font of the items in the pop-up menu.

MENUTITLE   The title of the pop-up menu. The default title is the label of the STATE item.

NWAY   [Free Menu Item]

NWAY items provide a way to collect any number of items together, in any format within the Free Menu. Only one item from each Collection can be selected at a time, and that item is highlighted to indicate this. The following Item Properties are particular to NWAY items:

COLLECTION   An identifier that specifies which NWAY Collection this item belongs to.

NWAYPROPS   A property list of information to be associated with this collection. This property is only noticed in the Free Menu Description on the first item in a COLLECTION. NWAY Collections are formed by creating a number of NWAY items with the same COLLECTION property. Each NWAY item acts individually as a Toggle item, and can have its own mouse event functions. Each NWAY Collection itself has properties, its state for instance. After the Free Menu is created, these Collection properties can be accessed by the macro FM.NWAYPROPS. Note that NWAY Collections are different from Free Menu Groups. There are three NWAY Collection properties that Free Menu looks at:

DESELECT   If given, specifies that the Collection can be deselected, yielding a state in which no item in the Collection is selected. When this property is set, the Collection can be deselected by selecting any item in the Collection and pressing the right mouse button .

STATE   The current state of the Collection, which is the actual item selected.

INITSTATE   Specifies the initial state of the Collection. The value of this property is an Item Link Description

EDIT   [Free Menu Item]

EDIT items are textual items that can be edited. The label for an EDIT item cannot be a bitmap. When the item is selected an edit caret appears at that cursor position within the item, allowing insertion and deletion of characters at that point. If selected with the right mouse button, the item is cleared before editing starts. While editing, the left mouse button moves the caret to a new position within the item. The right mouse button deletes from the caret to the cursor. CONTROL-W deletes the previous word. Editing is stopped when another item is selected, when the user moves the cursor into another TTY window and clicks the cursor, or when the Free Menu function FM.ENDEDIT is called (called when the Free Menu is reset, or the window is closed). The Free Menu editor will time out after about a minute, returning automatically. Because of the many ways in which editing can terminate, EDIT items are not allowed to have a SELECTEDFN, as it is not clear when this function should be called. Each EDIT item should have an ID specified, which is used when getting the state of the Free Menu, since the string being edited is defined as the state of the item, and thus cannot distinguish edit items. The following Item Properties are specific to EDIT items.

MAXWIDTH | Specifies the maximum string width of the item, in bits, after which input will be ignored. If `MAXWIDTH` is not specified, the items becomes infinitely wide and input is never restricted.

INFINITEWIDTH | This property is set automatically when `MAXWIDTH` is not specified. This tells Free Menu that the item has no right end, so that the item becomes mouse sensitive from its left edge to the right edge of the window, within the vertical space of the item.

In Medley, Changestate of an infinite width Edit item to a smaller item clears the old item properly.

LIMITCHARS | The input characters allowed can be restricted in two ways: If this item property is a list, it is treated as a list of legal characters; any character not in the list will be ignored. If it is an atom, it is treated as the name of a test predicate, which is passed three arguments, `ITEM`, `WINDOW`, and `CHARACTER`, when each character is typed. This predicate should return `T` if the character is legal, `NIL` otherwise. The `LIMITCHARS` function can also call `FM.ENDEDIT` to force the editor to terminate, or `FM.SKIPNEXT`, to cause the editor to jump to the next edit item in the menu.

ECHOCHAR | This item property can be set to any character. This character will be echoed in the window, regardless of what character is typed. However the item's label contains the actual string typed. This is useful for operations like password prompting. If `ECHOCHAR` is used, the font of the item must be fixed pitch. Unrestricted `EDIT` items should not have other items to their right in the menu, as they will be replaced. If the item is boxed, input is restricted to what will fit in the box. Typing off the edge of the window will cause the window to scroll appropriately. Control characters can be edited, including the carriage return and line feed, and they are echoed as a black box. While editing, the Skip/Next key ends editing the current item, and starts editing the next `EDIT` item in the Free Menu.

NUMBER [Free Menu Item]

`NUMBER` items are `EDIT` items that are restricted to numerals. The state of the item is coerced to the the number itself, not a string of numerals. There is one `NUMBER`- specific Item Property:

NUMBERTYPE | If `FLOATP` (or `FLOAT`), then decimals are accepted. Otherwise only whole numbers can be edited.

EDITSTART [Free Menu Item]

`EDITSTART` items serve the purpose of starting editing on another item when they are selected. The associated Edit item is linked to the EditStart item by an `EDIT` link (see Free Menu Item Links below). If the `EDITSTART` item is selected with the right mouse button, the Edit item is cleared before editing is started. Similar to `EDIT` items, `EDITSTART` items cannot have a `SELECTEDFN`, as it is not clear when the associated editing will terminate.

In Medley, `EDITSTART` items linked to a Number item properly set number state when editing has completed.

`DISPLAY` [Free Menu Item]

`DISPLAY` items serve two purposes. First, they simply provide a way of putting dummy text in a Free Menu, which does nothing when selected. The item's label can be changed, though. Secondly, `DISPLAY` items can be used as the base for new item types. The application can create new item types by specifying `DOWNFN`, `HELDFN`, `MOVEDFN`, and `SELECTEDFN` for a `DISPLAY` item, making it behave as desired.

Free Menu Item Highlighting

Each Free Menu Item can specify how it wants to be highlighted. First of all, if the item does not specify a `HIGHLIGHT` property, there are two default highlights. If the item is not boxed, the label is simply inverted, as in normal menus. If the item is boxed, it is highlighted in the shade of the box. Alternatively, the value of the `HIGHLIGHT` property can be a SHADE, which will be painted on top of the item when a mouse event occurs on it. Or the `HIGHLIGHT` property can be an alternate label, which can be an atom, string or bitmap. If the highlight label is a different size than the item label, the formatter will leave enough space for the larger of the two. In all of these cases, the looks of the highlighted item are determined when the Free Menu is built, and a bitmap of the item with these looks is created. This bitmap is stored on the item's `HIGHLIGHT` property, and simply displayed when a mouse event occurs. The value of the highlight property in the Item Description is copied to the `USERDATA` list, in case it is needed later for a label change.

Free Menu Item Links

Links between items are useful for grouping items in abstract ways. In particular, links are used for associating `EDITSTART` items with their item to edit, and `STATE` items with their state display. The Free Menu Item property `LINKS` is a property list, where the value of each Link Name property is a pointer to another item. In the Item Description, the value of the `LINK` property should be a property list as above. The value of each Link Name property is a Link Description. A Link Description can be one of the following forms:

<ID> An ID of an item in the Free Menu. This is acceptable if items can be distinguished by ID alone.

(<GROUPID> <ID>) A list whose first element is a GROUPID, and whose second element is the ID of an item in that group. This way items with similar purposes, and thus similar ID's, can be distinguished across groups.

(GROUP <ID>) A list whose first element is the keyword GROUP, and whose second element is an item ID. This form describes an item with ID, in the same group that this item is in. This way you do not need to know the GROUPID, just which group it is in.

Then after the entire menu is built, the links are set up, turning the Link Descriptions into actual pointers to Free Menu Items. There is no reason why circular Item Links cannot be created, although such a link would probably not be very useful. If circular links are created, the Free Menu will not be garbage collected after it is not longer being used. The application is responsible for breaking any such links that it creates.

Free Menu Window Properties

FM.PROMPTWINDOW   Specifies the window that Free Menu should use for displaying the item's messages. If not specified, PROMPTWINDOW is used.

FM.BACKGROUND   The background shade of the entire Free Menu. This property can be set automatically by specifying a BACKGROUND argument to the function FREEMENU. The window border must be 4 or greater when a Free Menu background is used, due to the way the Window System handles window borders.

FM.DONTRESHAPE   Normally, Free Menu will attempt to use empty space in a window by pushing items around to fill the space. When a Free Menu window is reshaped, the items are repositioned in the new shape. This can be disabled by setting the FM.DONTRESHAPE window property.

Free Menu Interface Functions

(**FREEMENU** *DESCRIPTION TITLE BACKGROUND BORDER*)   [Function]

Creates a Free Menu from a Free Menu Description, returning the window. This function will return quickly unless new display fonts have to be created.

Accessing Functions

(**FM.GETITEM** *ID GROUP WINDOW*)   [Function]

Gets item *ID* in *GROUP* of the Free Menu in *WINDOW*. This function will search the Free Menu for an item whose *ID* property matches, or secondly whose LABEL property matches *ID*. If *GROUP* is NIL, then the entire Free Menu is searched. If no matching item is found, NIL is returned.

(**FM.GETSTATE** *WINDOW*)   [Function]

Returns in property list format the ID and current STATE of every NWAY Collection and item in the Free Menu. If an item's or Collection's state is NIL, then it is not included in the list. This provides an easy way of getting the state of the menu all at once. If the state of only one item or Collection is needed, the application can directly access the STATE property of that object using the Accessing Macros described in Section 28.7.20, Free Menu Macros. This function can be called when editing is in progress, in which case it will provide the label of the item being edited at that point.

Changing Free Menus

Many of the following functions operate on Free Menu Items, and thus take the item as an argument. The *ITEM* argument to these functions can be the Free Menu Item itself, or just a reference to the item. In the second case, FM.GETITEM (see Section 28.7.16, Accessing Functions) will be used to find the item in the Free Menu. The reference can be in one of the following forms:

<ID>   Specifies the first item in the Free Menu whose ID or LABEL property matches <ID>.

(<GROUPID> <ID>)   Specifies the item whose ID or LABEL property matches <ID> within the group specified by <GROUPID>.

(**FM.CHANGELABEL** *ITEM NEWLABEL WINDOW UPDATEFLG*)   [Function]

Changes an *ITEM*'s label after the Free Menu has been created. It works for any type of item, and STATE items will remain in their current state. If the window is open, the item will be redisplayed with its new appearance. *NEWLABEL* can be an atom, a string, or a bitmap (except for EDIT items), and will be restricted in size by the MAXWIDTH and MAXHEIGHT Item Properties. If these properties are unspecified, the *ITEM* will be able to grow to any size. *UPDATEFLG* specifies whether or not the regions of the groups in the menu are recalculated to take into account the change of size of this item. The application should not change the label of an EDIT item while it is being edited. The following Item Property is relevant to changing labels:

CHANGELABELUPDATE   Exactly like UPDATEFLG except specified on the item, rather than as a function paramater.

(**FM.CHANGESTATE** *X NEWSTATE WINDOW*)   [Function]

Programmatically changes the state of items and NWAY Collections. *X* is either an item or a Collection name. For items *NEWSTATE* is a state appropriate to the type of the item. For NWAY Collections, *NEWSTATE* should be the desired item in the Collection, or NIL to deselect. For EDIT and NUMBER items, this function just does a label change. If the window is open, the item will be redisplayed.

(**FM.RESETSTATE** *ITEM WINDOW*)   [Function]

Sets an *ITEM* back to its initial state.

(**FM.RESETMENU** *WINDOW*)   [Function]

Resets every item in the menu back to its initial state.

(**FM.RESETSHAPE** *WINDOW ALWAYSFLG*)   [Function]

Reshapes the *WINDOW* to its full extent, leaving the lower-left corner unmoved. Unless *ALWAYSFLG* is T, the window will only be increased in size as a result of resetting the shape.

(**FM.RESETGROUPS** *WINDOW*)   [Function]

Recalculates the extent of each group in the menu, updating group boxes and backgrounds appropriately.

(**FM.HIGHLIGHTITEM** *ITEM WINDOW*) [Function]

Programmatically forces an *ITEM* to be highlighted. This might be useful for *ITEM*s which have a direct effect on other *ITEM*s in the menu. The *ITEM* will be highlighted according to its HIGHLIGHT property, as described in Section 28.7.12, Free Menu Item Highlighting. This highlight is temporary, and will be lost if the *ITEM* is redisplayed, by scrolling for example.

Editor Functions

(**FM.EDITITEM** *ITEM WINDOW CLEARFLG*) [Function]

Starts editing an EDIT or NUMBER *ITEM* at the beginning of the *ITEM*, as long as the *WINDOW* is open. This function will most likely be useful for starting editing of an *ITEM* that is currently the null string. If *CLEARFLG* is set, the *ITEM* is cleared first.

(**FM.SKIPNEXT** *WINDOW CLEARFLG*) [Function]

Causes the editor to jump to the beginning of the next EDIT item in the Free Menu. If *CLEARFLG* is set, then the next item will be cleared first. If there is not another EDIT item in the menu, this function will simply cause editing to stop. If this function is called when editing is not in progress, editing will begin on the first EDIT item in the menu. This function can be called from any process, and can also be called from inside the editor, in a LIMITCHARS function.

(**FM.ENDEDIT** *WINDOW WAITFLG*) [Function]

Stops any editing going on in *WINDOW*. If *WAITFLG* is T, then block until the editor has completely finished. This function can be called from another process, or from a LIMITCHARS function.

(**FM.EDITP** *WINDOW*) [Function]

If an item is in the process of being edited in the Free Menu *WINDOW*, that item is returned. Otherwise, NIL is returned.

Miscellaneous Functions

(**FM.REDISPLAYMENU** *WINDOW*) [Function]

Redisplays the entire Free Menu in its *WINDOW*, if the *WINDOW* is open.

(**FM.REDISPLAYITEM** *ITEM WINDOW*) [Function]

Redisplays a particular Free Menu *ITEM* in its *WINDOW*, if the *WINDOW* is open.

(**FM.SHADE** *X SHADE WINDOW*)   [Function]

> *X* can be an item, or a group ID. *SHADE* is painted on top of the item or group. Note that this is a temporary operation, and will be undone by redisplaying. For more permanent shading, the application may be able to add a REDEDISPLAYFN and SCROLLFN for the window as necessary to update the shading.

(**FM.WHICHITEM** *WINDOW POSorX Y*)   [Function]

> Locates and identifies an item from its known location within the *WINDOW*. If *WINDOW* is NIL, (WHICHW) is used, and if *POSorX* is NIL, the current cursor location is used.

(**FM.TOPGROUPID** *WINDOW*)   [Function]

> Returns the ID of the top group of this Free Menu.

Free Menu Macros

These Accessing Macros are provided to allow the application to get and set information in the Free Menu data structures. They are implemented as macros so that the operation will compile into the actual access form, rather than figuring that out at run time.

(**FM.ITEMPROP** *ITEM PROP {VALUE}*)   [Macro]

> Similar to WINDOWPROP, this macro provides an easy access to the fields of a Free Menu Item. The function FM.GETITEM gets the *ITEM*, described in Section 28.7.16, Accessing Function. *VALUE* is optional, and if not given, the current value of the *PROP* property will be returned. If *VALUE* is given, it will be used as the new value for that *PROP*, and the old value will be returned. When a call to FM.ITEMPROP is compiled, if the *PROP* is known (quoted in the calling form), the macro figures out what field to access, and the appropriate Data Type access form is compiled. However, if the *PROP* is not known at compile time, the function FM.ITEMPROP, which goes through the necessary property selection at run time, is compiled. The TYPE and USERDATA properties of a Free Menu Item are Read Only, and an error will result from trying to change the value of one of these properties.

(**FM.GROUPPROP** *WINDOW GROUP PROP {VALUE}*)   [Macro]

> Provides access to the Group Properties set up in the *PROPS* list for each group in the Free Menu Description. *GROUP* specifies the ID of the desired group, and *PROP* the name of the desired property. If *VALUE* is specified, it will become the new value of the property, and the old value will be returned. Otherwise, the current value is returned.

(**FM.MENUPROP** *WINDOW PROP {VALUE}*)   [Macro]

> Provides access to the group properties of the top-most group in the Free Menu, that is to say, the entire menu. This provides an easy way for the application to attach properties to the menu as a whole, as well as access the Group Properties for the entire menu.

(**FM.NWAYPROP** *WINDOW COLLECTION PROP {VALUE}*)   [Macro]

This macro works just like FM.GROUPPROP, except it provides access to the NWay Collections.

Attached Windows

The attached window facility makes it easy to manipulate a group of window as a unit. Standard window operations like moving, reshaping, opening, and closing can be done so that it appears to the user as if the windows are a single entity. Each collection of attached windows has one main window and any number of other windows that are "attached" to it. Moving or reshaping the main window causes all of the attached windows to be moved or reshaped as well. Moving or reshaping an attached window does not affect the main window.

Attached windows can have other windows attached to them. Thus, it is possible to attach window A to window B when B is already attached to window C. Similarly, if A has other windows attached to it, it can still be attached to B.

(**ATTACHWINDOW** *WINDOWTOATTACH MAINWINDOW EDGE POSITIONONEDGE WINDOWCOMACTION*)   [Function]

Associates *WINDOWTOATTACH* with *MAINWINDOW* so that window operations done to *MAINWINDOW* are also done to *WINDOWTOATTACH* (the exact set of window operations passed between main windows and attached windows is described in the Window Operations and Attached Windows section below). ATTACHWINDOW moves *WINDOWTOATTACH* to the correct position relative to *MAINWINDOW*.

Note: A window can be attached to only one other window. Attaching a window to a second window will detach it from the first. Attachments can not form loops. That is, a window cannot be attached to itself or to a window that is attached to it. ATTACHWINDOW will generate an error if this is attempted.

EDGE determines which edge of *MAINWINDOW* the attached window is positioned along: it should be one of TOP, BOTTOM, LEFT, or RIGHT. If *EDGE* is NIL, it defaults to TOP.

POSITIONONEDGE determines where along *EDGE* the attached window is positioned. It should be one of the following:

LEFT   The attached window is placed on the left (of a TOP or BOTTOM edge).

RIGHT   The attached window is placed on the right (of a TOP or BOTTOM edge).

BOTTOM   The attached window is placed on the bottom (of a LEFT or RIGHT edge).

TOP   The attached window is placed on the top (of a LEFT or RIGHT edge).

CENTER   The attached window is placed in the center of the edge.

```
JUSTIFY
```

or `NIL`   The attached window is placed to fill the entire edge. `ATTACHWINDOW` reshapes the window if necessary.

Note: The width or height used to justify an attached window includes any other windows that have already been attached to *MAINWINDOW*. Thus `(ATTACHWINDOW BBB AAA 'RIGHT 'JUSTIFY)` followed by `(ATTACHWINDOW CCC AAA 'TOP 'JUSTIFY)` will put CCC across the top of both BBB and AAA:

WINDOWCOMACTION provides a convenient way of specifying how *WINDOWTOATTACH* responds to right button menu commands. The window property `PASSTOMAINCOMS` determines which right button menu commands are directly applied to the attached window, and which are passed to the main window (see the Window Operations and Attached Windows section below). Depending on the value of *WINDOWCOMACTION*, the `PASSTOMAINCOMS` window property of *WINDOWTOATTACH* is set as follows:

`NIL`   `PASSTOMAINCOMS` is set to `(CLOSEW MOVEW SHAPEW SHRINKW BURYW)`, so right button menu commands to close, move, shape, shrink, and bury are passed to the main window, and all others are applied to the attached window.

`LOCALCLOSE`   `PASSTOMAINCOMS` is set to `(MOVEW SHAPEW SHRINKW BURYW)`, which is the same as when *WINDOWCOMACTION* is NIL, except that the attached window can be closed independently.

`HERE`   `PASSTOMAINCOMS` is set to `NIL`, so all right button menu commands are applied to the attached window.

`MAIN`   `PASSTOMAINCOMS` is set to `T`, so all right button menu commands are passed to the main window.

Note: If the user wants to set the `PASSTOMAINCOMS` window property of an attached window to something else, it must be done after the window is attached, since `ATTACHWINDOW` modifies this window property.

(**DETACHWINDOW** *WINDOWTODETACH*)   [Function]

Detaches *WINDOWTODETACH* from its main window. Returns a dotted pair (EDGE . POSITIONONEDGE) if *WINDOWTODETACH* was an attached window, NIL otherwise. This does not close *WINDOWTODETACH*.

(**DETACHALLWINDOWS** *MAINWINDOW*)   [Function]

Detaches and closes all windows attached to *MAINWINDOW*.

(**FREEATTACHEDWINDOW** *WINDOW*)   [Function]

Detaches the attached window *WINDOW*. In addition, other attached windows above (in the case of a TOP attached window) or below (in the case of a BOTTOM attached window) are moved closer to the main window to fill the gap.

> Note: Attached windows that "reject" the move operation (see REJECTMAINCOMS below) are not moved.

> Note: FREEATTACHEDWINDOW currently doesn't handle LEFT or RIGHT attached windows.

(**REMOVEWINDOW** *WINDOW*)   [Function]

Closes *WINDOW*, and calls FREEATTACHEDWINDOW to move other attached windows to fill any gaps.

(**REPOSITIONATTACHEDWINDOWS** *WINDOW*)   [Function]

Repositions every window attached to *WINDOW*, in the order that they were attached. This is useful as a RESHAPEFN for main windows with attached window that don't want to be reshaped, but do want to keep their position relative to the main window when the main window is reshaped.

> Note: Attached windows that "reject" the move operation (see REJECTMAINCOMS below) are not moved.

(**MAINWINDOW** *WINDOW RECURSEFLG*)   [Function]

If *WINDOW* is not a window, it generates an error. If *WINDOW* is closed, it returns *WINDOW*. If *WINDOW* is not attached to another window, it returns *WINDOW* itself. If *RECURSEFLG* is NIL and *WINDOW* is attached to a window, it returns that window. If *RECURSEFLG* is T, it returns the first window up the "main window" chain starting at *WINDOW* that is not attached to any other window.

(**ATTACHEDWINDOWS** *WINDOW COM*)   [Function]

Returns the list of windows attached to *WINDOW*.

If *COM* is non-NIL, only those windows attached to *WINDOW* that do not reject the window operation *COM* are returned (see REJECTMAINCOMS).

(**ALLATTACHEDWINDOWS** *WINDOW*)   [Function]

Returns a list of all of the windows attached to *WINDOW* or attached to a window attached to it.

(**WINDOWREGION** *WINDOW COM*)   [Function]

Returns the screen region occupied by *WINDOW* and its attached windows, if it has any.

If *COM* is non-NIL, only those windows attached to *WINDOW* that do not reject the window operation *COM* are considered in the calculation (see REJECTMAINCOMS).

(**WINDOWSIZE** *WINDOW*)   [Function]

Returns the size of *WINDOW* and its attached windows (if any), as a dotted pair (WIDTH . HEIGHT).

(**MINATTACHEDWINDOWEXTENT** *WINDOW*)   [Function]

Returns the minimum size that *WINDOW* and its attached windows (if any) will accept, as a dotted pair (WIDTH . HEIGHT).

Attaching Menus To Windows

The following functions are provided to associate menus to windows.

(**MENUWINDOW** *MENU VERTFLG*)   [Function]

Returns a closed window that has the menu *MENU* in it. If *MENU* is a list, a menu is created with *MENU* as its ITEMS menu field. Otherwise, *MENU* should be a menu. The returned window has the appropriate RESHAPEFN, MINSIZE and MAXSIZE window properties to allow its use in a window group.

If both the MENUROWS and MENUCOLUMNS fields of *MENU* are NIL, *VERTFLG* is used to set the default menu shape. If *VERTFLG* is non-NIL, the MENUCOLUMNS field of *MENU* will be set to 1 (the menu items will be listed vertically); otherwise the MENUROWS field of *MENU* will be set to 1 (the menu items will be listed horizontally).

(**ATTACHMENU** *MENU MAINWINDOW EDGE POSITIONONEDGE NOOPENFLG*)   [Function]

Creates a window that contains the menu *MENU* (by calling *MENUWINDOW*) and attaches it to the window *MAINWINDOW* on edge *EDGE* at position *POSITIONONEDGE*. The menu window is opened unless *MAINWINDOW* is closed, or *NOOPENFLG* is T.

If EDGE is either LEFT or RIGHT, MENUWINDOW will be called with VERTFLG = T, so the menu items will be listed vertically; otherwise the menu items will be listed horizontally. These defaults can be overridden by specifying the MENUROWS or MENUCOLUMNS fields in *MENU*.

(**CREATEMENUEDWINDOW** *MENU WINDOWTITLE LOCATION WINDOWSPEC*)   [Function]

INTERLISP-D REFERENCE MANUAL

Creates a window with an attached menu and returns the main window. *MENU* is the only required argument, and may be a menu or a list of menu items. *WINDOWTITLE* is a string specifying the title of the main window. *LOCATION* specifies the edge on which to place the menu; the default is TOP. *WINDOWSPEC* is a region specifying a region for the aggregate window; if NIL, the user is prompted for a region.

Examples:

```
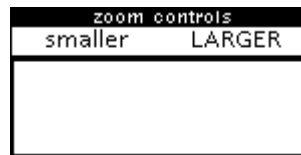(SETQ MENUW
   (MENUWINDOW
      (create MENU
         ITEMS ← '(smaller LARGER)
         MENUFONT ← '(MODERN 12)
         TITLE ← "zoom controls"
         CENTERFLG ← T
         WHENSELECTEDFN ← (FUNCTION ZOOMMAINWINDOW))))
```

creates (but does not open) a menu window that contains the two items "smaller" and "LARGER" with the title "zoom controls" and that calls the function ZOOMMAINWINDOW when an item is selected. Note that the menu items will be listed horizontally, because MENUWINDOW is called with VERTFLG = NIL, and the menu does not specify either a MENUROWS or MENUCOLUMNS field.

```
(ATTACHWINDOW MENUW
   (CREATEW '(50 50 150 50))
   'TOP
   'JUSTIFY)
```

creates a window on the screen and attaches the above created menu window to its top:


```
(CREATEMENUEDWINDOW
   (create MENU
      ITEMS ← '(smaller LARGER)
      MENUFONT ← '(MODERN 12)
      TITLE ← "zoom controls"
      CENTERFLG ← T
      WHENSELECTEDFN ← (FUNCTION ZOOMMAINWINDOW))))
```

creates the same sort of window in one step, prompting the user for a region.

Attached Prompt Windows

Many packages have a need to display status information or prompt for small amounts of user input in a place outside their standard window. A convenient way to do this is to attach a small window to the top of the program's main window. The following functions do so in a uniform way that can be depended on among diverse applications.

(**GETPROMPTWINDOW** *MAINWINDOW #LINES FONT DONTCREATE*)   [Function]

> Returns the attached prompt window associated with *MAINWINDOW*, creating it if necessary. The window is always attached to the top of *MAINWINDOW*, has DSPSCROLL set to T, and has a PAGEFULLFN of NILL to inhibit page holding. The window is at least *#LINES* lines high (default 1); if a pre-existing window is shorter than that, it is reshaped to make it large enough. *FONT* is the font to give the prompt window (defaults to the font of *MAINWINDOW*), and applies only when the window is first created. If *DONTCREATE* is true, returns the window if it exists, otherwise NIL without creating any prompt window.

(**REMOVEPROMPTWINDOW** *MAINWINDOW*)   [Function]

> Detaches the attached prompt window associated with *MAINWINDOW* (if any), and closes it.

Window Operations And Attached Windows

When a window operation, such as moving or clearing, is performed on a window, there is a question about whether or not that operation should also be performed on the windows attached to it or performed on the window it is attached to. The "right" thing to do depends on the window operation: it makes sense to independently redisplay a single window in a collection of windows, whereas moving a single window usually implies moving the whole group of windows. The interpretation of window operations also depends on the application that the window group is used for. For some applications, it may be desirable to have a window group where individual windows can be moved away from the group, but still be conceptually attached to the group for other operations. The attached window facility is flexible enough to allow all of these possibilities.

The operation of window operations can be specified by each attached window, by setting the following two window properties:

PASSTOMAINCOMS   [Window Property]

> Value is a list of window commands (e.g. CLOSEW, MOVEW) which, when selected from the attached window's right-button menu, are actually applied to the central window in the group, instead of being applied to the attached window itself. The "central window" is the first window up the "main window" chain that is not attached to any other window.

> If PASSTOMAINCOMS is NIL, all window operations are directly applied to the attached window. If PASSTOMAINCOMS is T, all window operations are passed to the central window.

> Note: ATTACHWINDOW allows this window property to be set to commonly-used values by using its WINDOWCOMACTION argument. ATTACHWINDOW always sets this window property, so users must modify it directly only after attaching the window to another window.

REJECTMAINCOMS   [Window Property]

Value is a list of window commands that the attached window will not allow the main window to apply to it. This is how a window can say "leave me out of this group operation."

If `REJECTMAINCOMS` is `NIL`, all window commands may be applied to this attached window. If `REJECTMAINCOMS` is `T`, no window commands may be applied to this attached window.

The `PASSTOMAINCOMS` and `REJECTMAINCOMS` window properties affect right-button menu operations applied to main windows or attached windows, and the action of programmatic window functions (`SHAPEW`, `MOVEW`, etc.) applied to main windows. However, these window properties do not affect the action of window functions applied to attached windows.

The following list describes the behavior of main and attached windows under the window operations, assuming that all attached windows have their `REJECTMAINCOMS` window property set to `NIL` and `PASSTOMAINCOMS` set to (`CLOSEW MOVEW SHAPEW SHRINKW BURYW`) (the default if `ATTACHWINDOW` is called with `WINDOWCOMACTION = NIL`).

The behavior for any particular operation can be changed for particular attached windows by setting the standard window properties (e.g., `MOVEFN` or `CLOSEFN`) of the attached window. An exception is the `TOTOPFN` property of an attached window, that is set to bring the whole window group to the top and should not be set by the user (although users can add functions to the `TOTOPFN` window property).

Move   If the main window moves, all attached windows move with it, and the relative positioning between the main window and the attached windows is maintained. If the region is determined interactively, the prompt region for the move is the union of the extent of the main window and all attached windows (excluding those with `MOVEW` in their `REJECTMAINCOMS` window property).

If an attached window is moved by calling the function `MOVEW`, it is moved without affecting the main window. If the right-button window menu command Move is called on an attached window, it is passed on to the main window, so that all windows in the group move.

Reshape   If the main window is reshaped, the minimum size of it and all of its attached windows is used as the minimum of the space for the result. Any space greater than the minimum is distributed among the main window and its attached windows. Attached windows with `SHAPEW` on their `REJECTMAINCOMS` window property are ignored when finding the minimum size, creating a "ghost" region, or distributing space after a reshape.

If an attached window is reshaped by calling the function `SHAPEW`, it is reshaped independently. If the right-button window menu command Shape is called on an attached window, it is passed on to the main window, so the whole group is reshaped.

Note: Reshaping the main window will restore the conditions established by the call to `ATTACHWINDOW`, whereas moving the main window does not. Thus, if A is attached to the top of B and then moved by the user, its new position relative to B will be maintained if B is moved. If B is reshaped, A will be reshaped to the top of B. Additionally, if, while A is moved away from the top of B, C is attached to the top of B, C will position itself above where A used to be.

Close If the main window is closed, all of the attached windows are closed also and the links from the attached windows to the main window are broken. This is necessary for the windows to be garbage collected.

If an attached window is closed by calling the function `CLOSEW`, it is closed without affecting the main window. If the right-button window menu command `Close` is called on an attached window, it is passed on to the main window. Note that closing an attached window detaches it.

Open If the main window is opened, it opens all attached windows and reestablishes links from them to the main window.

Attached windows can be opened independently and this does not affect the main window. Note that it is possible to reopen a closed attached window and not have it linked to its main window.

Shrink The collection of windows shrinks as a group. The `SHRINKFN`s of the attached windows are evaluated but the only icon displayed is the one for the main window.

Redisplay The main or attached windows can be redisplayed independently.

Totop If any main or attached window is brought to the top, all of the other windows are brought to the top also.

Expand Expanding any of the windows expands the whole collection.

Scrolling All of the windows involved in the group scroll independently.

Clear All windows clear independently of each other.

Window Properties Of Attached Windows

Windows that are involved in a collection either as a main window or as an attached window have properties stored on them. The only properties that are intended to be set be set by the user are the `MINSIZE`, `MAXSIZE`, `PASSTOMAINCOMS`, and `REJECTMAINCOMS` window properties. The other properties should be considered read only.

MINSIZE [Window Property]
MAXSIZE [Window Property]

Each of these window properties should be a dotted pair `(WIDTH . HEIGHT)` or a function to apply to the window that returns a dotted pair. The numbers are used when the main window is reshaped. The `MINSIZE` is used to determine the size of the smallest region acceptable during reshaping. Any amount greater than the collective minimum is spread evenly among the windows until each reaches `MAXSIZE`. Any excess is given to the main window.

If you give the main window of an attached window group a `MINSIZE` or `MAXSIZE` property, its value is moved to the `MAINWINDOWMINSIZE` or `MAINWINDOWMAXSIZE` property, so that the main window can be given a size function that computes the minimum or maximum size of the entire group. Thus, if you want to change the main window's minimum or maximum size after attaching windows to it, you should change the `MAINWINDOWMINSIZE` or `MAINWINDOWMAXSIZE` property instead.

This doesn't address the hard problem of overlapping attached windows side to side, for example if window A was attached as [`TOP`, `LEFT`] and B as [`TOP`, `RIGHT`]. Currently, the attached window functions do not worry about the overlap.

The default `MAXSIZE` is `NIL`, which will let the region grow indefinitely.

MAINWINDOW [Window Property]

Pointer from attached windows to the main window of the group. This link is not available if the main window is closed. The function `MAINWINDOW` is the preferred way to access this property.

ATTACHEDWINDOWS [Window Property]

Pointer from a window to its attached windows. The function `ATTACHEDWINDOWS` is the preferred way to access this property.

WHEREATTACHED [Window Property]

For attached windows, a dotted pair (`EDGE . POSITIONONEDGE`) giving the edge and position on the edge that determine how the attached window is placed relative to its main window.

The `TOTOPFN` window property on attached windows and the properties `TOTOPFN`, `DOSHAPEFN`, `MOVEFN`, `CLOSEFN`, `OPENFN`, `SHRINKFN`, `EXPANDFN` and `CALCULATEREGIONFN` on main windows contain functions that implement the attached window manipulation facilities. Care should be used in modifying or replacing these properties.

Communication of Window Menu Commands between Attached Windows is dependent on the name of function used to implement the window command, e.g., `CLOSEW` implements `CLOSE` (refer to `PASSTOMAINCOMS` documentation under Attached Windows). Consequently, if an application intercepts a window command by changing `WHENSELECTEDFN` for an item in the WindowMenu (for example, to advise the application that a window is being closed), windows may not behave correctly when attached to other windows.

To get around this problem, the Medley release provides the variable `*attached-window-command-synonyms*`. This   variable is an alist, where each element is of the form (new-command-function-name . old-command-function-name).

For example, if an application redefines the WindowMenu to call my-close-window when `CLOSE` is selected, that application should:

```
(cl:push '(my-close-window . il:closew) il:*attached-window-
command-synonyms*)
```

in order to tell the attached window system that my-close-window is a synonym function for `CLOSEW`.

28. HARDCOPY FACILITIES

Interlisp-D includes facilities for generating hardcopy in "Interpress" format and "Press" format. Interpress is a file format used for communicating documents to Xerox Network System printers such as the Xerox 8044 and Xerox 5700. Press is a file format used for communicating documents to Xerox laser Xerographic printers known by the names "Dover", "Spruce", "Penguin", and "Raven". There are also library packages available for supporting other types of printer formats (4045, FX-80, C150, etc.). The hardcopy facilities are designed to allow the user to support new types of printers with minimal changes to the user interface.

Files can be in a number of formats, including Interpress files, plain text files, and formatted Tedit files. In order to print a file on a given printer, it is necessary to identify the format of the file, convert the file to a format that the printer can accept, and transmit it. Rather than require that the user explicitly determine file types and do the conversion, the Interlisp-D hardcopy functions generate Interpress or other format output depending on the appropriate choice for the designated printer. The hardcopy functions use the variables PRINTERTYPES and PRINTFILETYPES (described below) to determine the type of a file, how to convert it for a given printer, and how to send it. By changing these variables, the user can define other kinds of printers and print to them using the normal hardcopy functions.

(**SEND.FILE.TO.PRINTER** *FILE HOST PRINTOPTIONS*)   [Function]

> The function SEND.FILE.TO.PRINTER causes the file *FILE* to be sent to the printer *HOST*. If *HOST* is NIL, the first host in the list DEFAULTPRINTINGHOST which can print *FILE* is used.

> *PRINTOPTIONS* is a property list of the form (PROP$_1$ VALUE$_1$ PROP$_2$ VALUE$_2$ ...). The properties accepted depends on the type of printer. For Interpress printers, the following properties are accepted:

DOCUMENT.NAME   The document name to appear on the header page (a string). Default is the full name of the file.

DOCUMENT.CREATION.DATE   The creation date to appear on the header page (a Lisp integer date, such as returned by IDATE). The default value is the creation date of the file.

SENDER.NAME   The name of the sender to appear on the header page (a string). The default value is the name of the user.

RECIPIENT.NAME   The name of the recipient to appear on the header page (a string). The default is none.

MESSAGE   An additional message to appear on the header page (a string). The default is none.

#COPIES   The number of copies to be printed. The default value is 1.

PAGES.TO.PRINT   The pages of the document that should be printed, represented as a list `(FIRSTPAGE# LASTPAGE#)`. For example, if this option is (3 5), this specifies that pages 3 through 5, inclusive, should be printed. Note that the page numbering used for this purpose has no connection to any page numbers that may be printed on the document. The default is to print all of the pages in the document.

MEDIUM   The medium on which the master is to be printed. If omitted, this defaults to the value of `NSPRINT.DEFAULT.MEDIUM`, as follows: `NIL` means to use the printer's default; `T` means to use the first medium reported available by the printer; any other value must be a Courier value of type `MEDIUM`. The format of this type is a list `(PAPER (KNOWN.SIZE TYPE))` or `(PAPER (OTHER.SIZE (WIDTH LENGTH)))`. The paper `TYPE` is one of `US.LETTER`, `US.LEGAL`, `A0` through `A10`, `ISO.B0` through `ISO.B10`, and `JIS.B0` through `JIS.B10`. For users who use A4 paper exclusively, it should be sufficient to set `NSPRINT.DEFAULT.MEDIUM` to `(PAPER (KNOWN.SIZE "A4"))`.

When using different paper sizes, it may be necessary to reset the variable `DEFAULTPAGEREGION`, the region on the page used for printing (measured in micas from the lower-left corner).

STAPLE?   True if the document should be stapled.

#SIDES   1 or 2 to indicate that the document should be printed on one or two sides, respectively. The default is the value of `EMPRESS#SIDES`.

PRIORITY   The priority of this print request, one of `LOW`, `NORMAL`, or `HIGH`. The default is the printer's default.

Note: Press printers only recognize the options `#COPIES`, `#SIDES`, `DOCUMENT.CREATION.DATE`, and `DOCUMENT.NAME`.

For example,

```
(SEND.FILE.TO.PRINTER 'FOO NIL

    '(#COPIES 3 #SIDES 2 DOCUMENT.NAME "For John"))
```

`SEND.FILE.TO.PRINTER` calls `PRINTERTYPE` and `PRINTFILETYPE` to determine the printer type of *HOST* and the file format of *FILE*. If *FILE* is a formatted file already in a form that the printer can print, it is transmitted directly. Otherwise, `CONVERT.FILE.TO.TYPE.FOR.PRINTER` is called to do the conversion. [Note: If the file is converted, `PRINTOPTIONS` is passed to the formatting function, so it can include properties such as `HEADING`, `REGION`, and `FONTS`.] All of these functions use the lists `PRINTERTYPES` and `PRINTFILETYPES` to actually determine how to do the conversion.

LISTFILES (Chapter 17) calls the function LISTFILES1 to send a single file to a hardcopy printing device. Interlisp-D is initialized with LISTFILES1 defined to call SEND.FILE.TO.PRINTER.

(**HARDCOPYW** *WINDOW/BITMAP/REGION FILE HOST SCALEFACTOR ROTATION PRINTERTYPE HARDCOPYTITLE*) [Function]

Creates a hardcopy file from a bitmap and optionally sends it to a printer. Note that some printers may have limitations concerning how big or how "complicated" the bitmap may be printed.

WINDOW/BITMAP/REGION can either be a *WINDOW* (open or closed), a *BITMAP*, or a *REGION* (interpreted as a region of the screen). If *WINDOW/BITMAP/REGION* is NIL, the user is prompted for a screen region using GETREGION.

If *FILE* is non-NIL, it is used as the name of the file for output. If *HOST* = NIL, this file is not printed. If *FILE* is NIL, a temporary file is created, and sent to *HOST*.

To save an image on a file without printing it, perform (HARDCOPYW IMAGE FILE). To print an image to the printer PRINTER without saving the file, perform (HARDCOPYW IMAGE NIL PRINTER).

If both *FILE* and *HOST* are NIL, the default action is to print the image, without saving the file. The printer used is determined by the argument PRINTERTYPE and the value of the variable DEFAULTPRINTINGHOST. If PRINTERTYPE is non-NIL, the first host on DEFAULTPRINTINGHOST of the type PRINTERTYPE is used. If PRINTERTYPE is NIL, the first printer on DEFAULTPRINTINGHOST that implements the BITMAPSCALE (as determined by PRINTERTYPES) operation is used, if any. Otherwise, the first printer on DEFAULTPRINTINGHOST is used.

The type of hardcopy file produced is determined by HOST if non-NIL, else by PRINTERTYPE if non-NIL, else by the value of DEFAULTPRINTINGHOST, as described above.

SCALEFACTOR is a reduction factor. If not given, it is computed automatically based on the size of the bitmap and the capabilities of the printer type. This may not be supported for some printers.

ROTATION specifies how the bitmap image should be rotated on the printed page. Most printers (including Interpress printers) only support a ROTATION of multiples of 90.

PRINTERTYPE specifies what type of printer to use when HOST is NIL. HARDCOPYW uses this information to select which printer to use or what print file format to convert the output into, as described above.

The background menu contains a "Hardcopy" command (Chapter 28) that prompts the user for a region on the screen, and sends the image to the default printer.

Hardcopy output may also be obtained by writing a file on the printer device LPT, e.g. (COPYFILE 'FOO '{LPT}). When a file on this device is closed, it is converted to Interpress or some other format (if necessary) and sent to the default printer (the first host

on `DEFAULTPRINTINGHOST`). One can include the printer name directly in the file name, e.g. (`COPYFILE 'FOO {LPT}TREMOR:`) will send the file to the printer `TREMOR:`.

HARDCOPYTITLE is a string specifying a title to print on the page containing the screen image. If `NIL`, the string "Window Image" is used. To omit a title, specify the null string.

(**PRINTERSTATUS** *PRINTER*)   [Function]

Returns a list describing the current status of the printer named *PRINTER*. The exact form of the value returned depends on the type of printer. For InterPress printers, the status describes whether the printer is available or busy or needs attention, and what type of paper is loaded in the printer.

Returns `NIL` if the printer does not respond in a reasonable time, which can occur if the printer is very busy, or does not implement the printer status service.

DEFAULTPRINTINGHOST   [Variable]

The variable `DEFAULTPRINTINGHOST` is used to designate the default printer to be used as the output of printing operations. It should be a list of the known printer host names, for example, (`QUAKE LISPPRINT:`). If an element of `DEFAULTPRINTINGHOST` is a list, is interpreted as (`PRINTERTYPE HOST`), specifying both the host type and the host name. The type of the printer, which determines the protocol used to send to it and the file format it requires, is determined by the function `PRINTERTYPE`.

If `DEFAULTPRINTINGHOST` is a single printer name, it is treated as if it were a list of one element.

(**PRINTFILETYPE** *FILE* —)   [Function]

Returns the format of the file *FILE*. Possible values include `INTERPRESS`, `TEDIT`, etc. If it cannot determine the file type, it returns `NIL`. Uses the global variable `PRINTFILETYPES`.

(**PRINTERTYPE** *HOST*)   [Function]

Returns the type of the printer *HOST*. Currently uses the following heuristic:

  1. If *HOST* is a list, the `CAR` is assumed to be the printer type and `CADR` the name of the printer

  2. If *HOST* is a litatom with a non-`NIL` `PRINTERTYPE` property, the property value is returned as the printer type

  3. If *HOST* contains a colon (e.g., `PRINTER:PARC:XEROX`) it is assumed to be an `INTERPRESS` printer

4. If *HOST* is the CADR of a list on DEFAULTPRINTINGHOST, the CAR is returned as the printer type

5. Otherwise, the value of DEFAULTPRINTERTYPE is returned as the printer type.

Low-level Hardcopy Variables

The following variables are used to define how Interlisp should generate hardcopy of different types. The user should only need to change these variables when it is necessary to access a new type of printer, or define a new hardcopy document type (not often).

PRINTERTYPES   [Variable]

The characteristics of a given printer are determined by the value of the list PRINTERTYPES. Each element is a list of the form

$$(\text{TYPES } (\text{PROPERTY}_1 \text{ VALUE}_1) \text{ } (\text{PROPERTY}_2 \text{ VALUE}_2)$$
$$\text{...})$$

TYPES is a list of the printer types that this entry addresses. The (PROPERTY$_n$ VALUE$_n$) pairs define properties associated with each printer type.

The printer properties include the following:

CANPRINT   Value is a list of the file types that the printer can print directly.

STATUS   Value is a function that knows how to find out the status of the printer, used by PRINTERSTATUS.

PROPERTIES   Value is a function which returns a list of known printer properties.

SEND   Value is a function which invokes the appropriate protocol to send a file to the printer.

BITMAPSCALE   Value is a function of arguments WIDTH and HEIGHT in bits which returns a scale factor for scaling a bitmap.

BITMAPFILE   Value is a form which, when evaluated, converts a bitmap to a file format that the printer will accept.

Note: The name 8044 is defined on PRINTERTYPES as a synonym for the INTERPRESS printer type. The names SPRUCE, PENGUIN, and DOVER are defined on PRINTERTYPES as synonyms for the PRESS printer type. The printer types FULLPRESS and RAVEN are also defined the same as PRESS, except that these printer types indicate that the printer is a "Full Press" printer that is able to scale bitmap images, in addition to the normal Press printer facilities.

PRINTFILETYPES [Variable]

The variable `PRINTFILETYPES` contains information about various file formats, such as Tedit files and Interpress files. The format is similar to `PRINTERTYPES`. The properties that can be specified include:

TEST   Value is a function which tests a file if it is of the given type. Note that this function is passed an open stream.

CONVERSION   Value is a property list of other file types and funcitons that convert from the specified type to the file format.

EXTENSION   Value is a list of possible file extensions for files of this type.

29. TERMINALINPUT/OUTPUT

Most input/output operations in Interlisp can be simply modeled as reading or writing on a linear stream of bytes. However, the situation is much more complex when it comes to controlling the user's "terminal," which includes the keyboard, the mouse, and the display screen. For example, Interlisp coordinates the operation of these separate I/O devices so that the cursor on the screen moves as the mouse moves, and any characters typed by the user appear in the window currently containing a flashing cursor. Most of the time, this system works correctly without need for user modification.

The purpose of this chapter is to describe how to access the low-level controls for the terminal I/O devices. It documents the use of interrupt characters, the keyboard characters that generate interrupts. Then, it describes terminal tables, used to determine the meaning of the different editing characters (character delete, line delete, etc.). Then, the "dribble file" facility that allows terminal I/O to be saved onto a file is presented (see the Dribble Files section below). Finally, the low-level functions that control the mouse and cursor, the keyboard, and the screen are documented.

Interrupt Characters

Errors and breaks can be caused by errors within functions, or by explicitly breaking a function. The user can also indicate his desire to go into a break while a program is running by typing certain control characters known as "interrupt characters". The following interrupt characters are currently enabled in Interlisp-D:

Note: In Interlisp-D with multiple processes, it is not sufficient to say that "the computation" is broken, aborted, etc; it is necessary to specify which process is being acted upon. Usually, the user wants interrupts to occur in the TTY process, which is the one currently receiving keyboard input. However, sometimes the user wants to interrupt the mouse process, if it is currently busy executing a menu command or waiting for the user to specify a region on the screen. Most of the interrupt characters below take place in the mouse process if it is busy, otherwise the TTY process. Control-G can be used to break arbitrary processes. For more information, see Chapter 23.

      Control-B   Causes a break within the mouse process (if busy) or the TTY process. Use Control-G to break a particular process.

      Control-D   Aborts the mouse process (if busy) or the TTY process, and unwinds its stack to the top level. Calls RESET (see Chapter 14).

      Control-E   Aborts the mouse process (if busy) or the TTY process, and unwinds its stack to the last ERRORSET. Calls ERROR! (see Chapter 14).

      Control-G   Pops up a menu listing all of the currently-running processes. Selecting one of the processes will cause a break to take place in that process.

      Control-P   This interrupt is no longer supported in Medley.

Control-T   Flashes the `TTY` process' window and prints status information for the `TTY` process. First it prints "**IO wait**," "**Waiting**", or "**Running**," depending on whether the `TTY` process is currently in waiting for characters to be typed, waiting for some other reason, or running. Next, it prints the names of the top three frames on the stack, to show what is running. Then, it prints a line describing the percentage of time (since the last control-T) that has been spent running a program, swapping, garbage collecting, doing local disk I/O, etc. For example:

Running in `TTWAITFORINPUT` in `TTBIN` in `TTYIN1`

95% Util, 0% Swap, 4% GC

DELETE   Clears typeahead in all processes.

The user can disable and/or redefine Interlisp interrupt characters, as well as define new interrupt characters. Interlisp-D is initialized with the following interrupt channels: RESET (**Control-D**), ERROR (**Control-E**), BREAK (**Control-B**), HELP (**Control-G**), PRINTLEVEL (**Control-P**), RUBOUT (**DELETE**), and RAID. Each of these channels independently can be disabled, or have a new interrupt character assigned to it via the function INTERRUPTCHAR described below. In addition, the user can enable new interrupt channels, and associate with each channel an interrupt character and an expression to be evaluated when that character is typed.

(**INTERRUPTCHAR** *CHAR TYP/FORM HARDFLG* −)   [Function]

Defines *CHAR* as an interrupt character. If *CHAR* was previously defined as an interrupt character, that interpretation is disabled.

CHAR is either a character or a character code (see Chapter 2). Note that full sixteen-bit NS characters can be specified as interrupt characters (see Chapter 2). *CHAR* can also be a value returned from INTERRUPTCHAR, as described below.

If *TYP/FORM* = NIL, *CHAR* is disabled.

If *TYP/FORM* = T, the current state of *CHAR* is returned without changing or disabling it.

If *TYP/FORM* is one of the literal atoms RESET, ERROR, BREAK, HELP, PRINTLEVEL, RUBOUT, or RAID, then INTERRUPTCHAR assigns *CHAR* to the indicated Interlisp interrupt channel, (reenabling the channel if previously disabled).

If the argument *TYP/FORM* is a symbol designating a predefined system interrupt (RESET, ERROR, BREAK, etc), and HARDFLG is omitted or NIL, then the hardness defaults to the standard hardness of the system interrupt (e.g., MOUSE for the ERROR interrupt).

If *TYP/FORM* is any other literal atom, *CHAR* is enabled as an interrupt character that when typed causes the atom *TYP/FORM* to be immediately set to T.

If *TYP/FORM* is a list, *CHAR* is enabled as a user interrupt character, and *TYP/FORM* is the form that is evaluated when *CHAR* is typed. The interrupt will be hard if *HARDFLG* = T, otherwise soft.

(INTERRUPTCHAR T) restores all Interlisp channels to their original state, and disables all user interrupts.

HARDFLG determines what process the interrupt should run in. If *HARDFLG* is NIL, the interrupt will run in the TTY process, which is the process currently receiving keyboard input. If *HARDFLG* is T, the interrupt will occur in whichever process happens to be running. If *HARDFLG* is MOUSE, the interrupt will happen in the mouse process, if the mouse is busy, otherwise in the TTY process.

INTERRUPTCHAR returns a value which, when given as the *CHAR* argument to INTERRUPTCHAR, will restore things as they were before the call to INTERRUPTCHAR. Therefore, INTERRUPTCHAR can be used in conjunction with RESETFORM or RESETLST (see Chapter 14).

INTERRUPTCHAR **is undoable.**

(**RESET.INTERRUPTS** *PERMITTEDINTERRUPTS SAVECURRENT?*)   [Function]

PERMITTEDINTERRUPTS is a list of interrupt character settings to be performed, each of the form (CHAR TYP/FORM HARDFLG). The effect of RESET.INTERRUPTS is as if (INTERRUPTCHAR CHAR TYP/FORM HARDFLG) were performed for each item on PERMITTEDINTERRUPTS, **and** (INTERRUPTCHAR OTHERCHAR NIL) were performed on every other existing interrupt character.

If *SAVECURRENT?* is non-NIL, then RESET.INTERRUPTS returns the current state of the interrupts in a form that could be passed to RESET.INTERRUPTS, otherwise it returns NIL. This can be used with a RESET.INTERRUPTS that appears in a RESETFORM, so that the list is built at "entry", but not upon "exit".

(**LISPINTERRUPTS**)   [Function]

Returns the initial default interrupt character settings for Interlisp-D, as a list that RESET.INTERRUPTS would accept.

(**INTERRUPTABLE** *FLAG*)   [Function]

if *FLAG* = NIL, turns interrupts off. If *FLAG* = T, turns interrupts on. Value is previous setting. INTERRUPTABLE compiles open.

Any interrupt character typed while interrupts are off is treated the same as any other character, i.e., placed in the input buffer, and will not cause an interrupt when interrupts are turned back on.

Terminal Tables

A read table (see Chapter 25) contains input/output information that is media-independent. For example, the action of parentheses is the same regardless of the device from which the input is being performed. A terminal table is an object that contains information that pertains to terminal input/output operations only, such as the character to type to delete the last character or to delete the last line. In addition, terminal tables contain such information as how line-buffering is to be performed, how control characters are to be echoed/printed, whether lowercase input is to be converted to upper case, etc.

Using the functions below, the user may change, reset, or copy terminal tables, or create a new terminal table and install it as the primary terminal table via SETTERMTABLE. However, unlike read tables, terminal tables cannot be passed as arguments to input/output functions.

(**GETTERMTABLE** *TTBL*) [Function]

> If *TTBL* = NIL, returns the primary (i.e., current) terminal table. If *TTBL* is a terminal table, return *TTBL*. Otherwise, generates an **ILLEGAL TERMINAL TABLE** error.

(**COPYTERMTABLE** *TTBL*) [Function]

> Returns a copy of *TTBL*. *TTBL* can be a real terminal table, NIL (copies the primary terminal table), or ORIG (returns a copy of the original system terminal table). Note that COPYTERMTABLE is the only function that creates a terminal table.

(**SETTERMTABLE** *TTBL*) [Function]

> Sets the primary terminal table to be *TTBL*. Returns the previous primary terminal table. Generates an **ILLEGAL TERMINAL TABLE** error if *TTBL* is not a real terminal table.

(**RESETTERMTABLE** *TTBL* *FROM*) [Function]

> Copies (smashes) *FROM* into *TTBL*. *FROM* and *TTBL* can be NIL or a real terminal table. In addition, *FROM* can be ORIG, meaning to use the system's original terminal table.

(**TERMTABLEP** *TTBL*) [Function]

> Returns *TTBL*, if *TTBL* is a real terminal table, NIL otherwise.

Terminal Syntax Classes

A terminal table associates with each character a single "terminal syntax class", one of CHARDELETE, LINEDELETE, WORDDELETE, RETYPE, CTRLV, EOL, and NONE. Unlike read table classes, only one character in a particular terminal table can belong to each of the classes (except for the default class NONE). When a new character is assigned one of these syntax classes by SETSYNTAX (see Chapter 25), the previous character is disabled (i.e., reassigned the syntax class NONE), and the value of SETSYNTAX is the code for the previous character of that class, if any, otherwise NIL.

The terminal syntax classes are interpreted as follows:

CHARDELETE   (Initially BackSpace and Control-A in Interlisp-D) Typing this character deletes the previous character typed. Repeated use of this character deletes successive characters back to the beginning of the line.

LINEDELETE   (Initially Control-Q in Interlisp-D) Typing this character deletes the whole line; it cannot be used repeatedly.

WORDDELETE   (Initially Control-W in Interlisp-D) Typing this character deletes the previous "word", i.e., sequence of non-separator characters.

RETYPE   (Initially Control-R) Causes the line to be retyped as Interlisp sees it (useful when repeated deletions make it difficult to see what remains).

CTRLV
CNTRLV   (Initially Control-V) When followed by A, B, ... Z, inputs the corresponding control character control-A, control-B, ... control-Z. This allows interrupt characters to be input without causing an interrupt.

EOL   On input from a terminal, the EOL character signals to the line buffering routine to pass the input back to the calling function. It also is used to terminate inputs to READLINE (see Chapter 13). In general, whenever the phrase carriage-return linefeed is used, what is meant is the character with terminal syntax class EOL.

NONE   The terminal syntax class of all other characters.

GETSYNTAX, SETSYNTAX, and SYNTAXP all work on terminal tables as well as read tables (see page X.XX). As with read tables, full sixteen-bit NS characters can be specified in terminal tables (see Chapter 2). When given NIL as a TABLE argument, GETSYNTAX and SYNTAXP use the primary read table or primary terminal table depending on which table contains the indicated CLASS argument. For example, (SETSYNTAX CH 'BREAK) refers to the primary read table, and (SETSYNTAX CH 'CHARDELETE) refers to the primary terminal table. In the absence of such information, all three functions default to the primary read table; e.g., (SETSYNTAX '{ '%[) refers to the primary read table. If given incompatible CLASS and table arguments, all three functions generate errors. For example, (SETSYNTAX CH 'BREAK TTBL), where TTBL is a terminal table, generates an **ILLEGAL READTABLE** error, and (GETSYNTAX 'CHARDELETE RDTBL) generates an **ILLEGAL TERMINAL TABLE** error.

Terminal Control Functions

(**ECHOCHAR** *CHARCODE MODE TTBL*)   [Function]

ECHOCHAR sets the "echo mode" of the character *CHARCODE* to *MODE* in the terminal table *TTBL*. The "echo mode" determines how the character is to be echoed or printed. Note that although the name of this function suggests echoing only, it affects all output of the character, both echoing of input and printing of output.

CHARCODE should be a character code. *CHARCODE* can also be a list of characters, in which case ECHOCHAR is applied to each of them with arguments *MODE* and *TTBL*. Note that echo modes can be specified for full sixteen-bit NS characters (see Chapter 2).

MODE should be one of the litatoms IGNORE, REAL, SIMULATE, or INDICATE which specify how the character should be echoed or printed:

IGNORE	*CHARCODE* is never printed.
REAL	*CHARCODE* itself is printed. Some terminals may respond to certain control and meta characters in interesting ways.
SIMULATE	Output of *CHARCODE* is simulated. For example, control-I (tab) may be simulated by printing spaces. The simulation is machine-specific and beyond the control of the user.
INDICATE	For control or meta characters, *CHARCODE* is printed as # and/or ↑ followed by the corresponding alphabetic character. For example, Control-A would echo as ↑A, and meta-Control-W would echo as #↑W.

The value of ECHOCHAR is the previous echo mode for *CHARCODE*. If *MODE* = NIL, ECHOCHAR returns the current echo mode without changing it.

Warning: In some fonts, control and meta characters may be used for printable characters. If the echomode is set to INDICATE for these characters, they will not print out correctly.

(**ECHOCONTROL** *CHAR MODE TTBL*) [Function]

ECHOCONTROL is an old, limited version of ECHOCHAR, that can only specify the echo mode of control characters. *CHAR* is a character or character code. If *CHAR* is an alphabetic character (or code), it refers to the corresponding control character, e.g., (ECHOCONTROL 'Z 'INDICATE) if equivalent to (ECHOCHAR (CHARCODE ↑Z) 'INDICATE).

(**ECHOMODE** *FLG TTBL*) [Function]

If *FLG* = T, turns echoing for terminal table *TTBL* on. If *FLG* = NIL, turns echoing off. Returns the previous setting.

Note: Unlike ECHOCHAR, this only affects echoing of typed-in characters, not printing of characters.

(**GETECHOMODE** *TTBL*) [Function]

Returns the current echo mode for *TTBL*.

The following functions manipulate the "raise mode," which determines whether lower case characters are converted to upper case when input from the terminal. There is no "raise mode" for input from files.

(**RAISE** *FLG TTBL*)   [Function]

> Sets the RAISE mode for terminal table *TTBL*. If *FLG* = NIL, all characters are passed as typed. If *FLG* = T, input is echoed as typed, but lowercase letters are converted to upper case. If *FLG* = 0, input is converted to uppercase before it is echoed. Returns the previous setting.

(**GETRAISE** *TTBL*)   [Function]

> Returns the current RAISE mode for *TTBL*.

(**DELETECONTROL** *TYPE MESSAGE TTBL*)   [Function]

> Specifies the output protocol when a CHARDELETE or LINEDELETE is typed, by specifying character strings to print when characters are deleted.

> Interlisp-10 (designed for use on hardcopy terminals) echos the characters being deleted, preceding the first by a \ and following the last by a \, so that it is easy to see exactly what was deleted. Interlisp-D is initially set up to physically erase the deleted characters from the display, so the DELETECONTROL strings are initialized to the null string.

> The various values of *TYPE* specify different phases of the deletion, as follows:

> | | |
> |---|---|
> | 1STCHDEL | *MESSAGE* is the message printed the first time CHARDELETE is typed. Initially "\" in Interlisp-10. |
> | NTHCHDEL | *MESSAGE* is the message printed when the second and subsequent CHARDELETE characters are typed (without intervening characters). Initially "" in Interlisp-10. |
> | POSTCHDEL | *MESSAGE* is the message printed when input is resumed following a sequence of one or more CHARDELETE characters. Initially "\" in Interlisp-10. |
> | EMPTYCHDEL | *MESSAGE* is the message printed when a CHARDELETE is typed and there are no characters in the buffer. Initially "##cr" in Interlisp-10. |
> | ECHO | If *TYPE* = ECHO, the characters deleted by CHARDELETE are echoed. *MESSAGE* is ignored. |
> | NOECHO | If *TYPE* = NOECHO, the characters deleted by CHARDELETE are not echoed. *MESSAGE* is ignored. |
> | LINEDELETE | *MESSAGE* is the message printed when the LINEDELETE character is typed. Initially "##cr". |

Note: In Interlisp-10, the `LINEDELETE`, `1STCHDEL`, `NTHCHDEL`, `POSTCHDEL`, and `EMPTYCHDEL` messages must be 4 characters or fewer in length.

`DELETECONTROL` returns the previous message as a string. If *MESSAGE* `=` `NIL`, the value returned is the previous message without changing it. For *TYPE* `=` `ECHO` and `NOECHO`, the value of `DELETECONTROL` is the previous echo mode, i.e., `ECHO` or `NOECHO`.

(**GETDELETECONTROL** *TYPE TTBL*)   [Function]

Returns the current `DELETECONTROL` mode for *TYPE* in *TTBL*.

Line-Buffering

Characters typed at the terminal are stored in two buffers before they are passed to an input function. All characters typed in are put into the low-level "system buffer", which allows type-ahead. When an input function is entered, characters are transferred to the "line buffer" until a character with terminal syntax class `EOL` appears (or, for calls from `READ`, when the count of unbalanced open parentheses reaches 0). Note that `PEEKC` is an exception; it returns the character immediately when its second argument is `NIL`. Until this time, the user can delete characters one at a time from the line buffer by typing the current `CHARDELETE` character, or delete the entire line buffer back to the last carriage-return by typing the current `LINEDELETE`.

This line editing is not performed by `READ` or `RATOM`, but by Interlisp, i.e., it does not matter (nor is it necessarily known) which function will ultimately process the characters, only that they are still in the Interlisp line buffer. However, the function that is requesting input at the time the buffering starts does determine whether parentheses counting is observed. For example, if a program performs (`PROGN` (`RATOM`) (`READ`)) and the user types in "A (B C D)", the user must type in the carriage-return following the right parenthesis before any action is taken, because the line buffering is happening under `RATOM`. If the program had performed (`PROGN` (`READ`) (`READ`)), the line-buffering would be under `READ`, so that the right parenthesis would terminate line buffering, and no terminating carriage-return would be required.

Once a carriage-return has been typed, the entire line is "available" even if not all of it is processed by the function initiating the request for input. If any characters are "left over", they are returned immediately on the next request for input. For example, (`LIST` (`RATOM`) (`READC`) (`RATOM`)) when the input is "A Bcr" returns the three-element list (A % B) and leaves the carriage-return in the buffer.

If a carriage-return is typed when the input under `READ` is not "complete" (the parentheses are not balanced or a string is in progress), line buffering continues, but the lines completed so far are not available for editing with `CHARDELETE` or `LINEDELETE`.

The function `CONTROL` is available to defeat line-buffering:

(**CONTROL** *MODE TTBL*)   [Function]

If *MODE* = T, eliminates Interlisp's normal line-buffering for the terminal table *TTBL*. If *MODE* = NIL, restores line-buffering (normal). When operating with a terminal table in which (CONTROL T) has been performed, characters are returned to the calling function without line-buffering as described below.

CONTROL returns its previous setting.

(**GETCONTROL** *TTBL*)   [Function]

Returns the current control mode for *TTBL*.

The function that initiates the request for input determines how the line is treated when (CONTROL T) is in effect:

READ   If the expression being typed is a list, the effect is the same as though done with (CONTROL NIL), i.e., line-buffering continues until a carriage-return or matching parentheses. If the expression being typed is not a list, it is returned as soon as a break or separator character is encountered, e.g., (READ) when the input is "ABC<space>" immediately returns ABC. CHARDELETE and LINEDELETE are available on those characters still in the buffer. Thus, if a program is performing several reads under (CONTROL T), and the user types "NOW IS THE TIME" followed by Control-Q, only TIME is deleted, since the rest of the line has already been transmitted to READ and processed.

An exception to the above occurs when the break or separator character is an opening parenthesis, bracket or double-quote, since returning at this point would leave the line buffer in a "funny" state. Thus if the input to (READ) is "ABC(", the ABC is not read until a carriage-return or matching parentheses is encountered. In this case the user could LINEDELETE the entire line, since all of the characters are still in the buffer.

RATOM   Characters are returned as soon as a break or separator character is encountered. Until then, LINEDELETE and CHARDELETE may be used as with READ. For example, (RATOM) followed by "ABC<control-A><space>" returns AB. (RATOM) followed by "(<control-A>" returns (and types ## indicating that control-A was attempted with nothing in the buffer, since the (is a break character and would therefore already have been read.

READC
PEEKC   The character is returned immediately; no line editing is possible. In particular, (READC) is perfectly happy to return

the CHARDELETE or LINEDELETE characters, or the ESCAPE character (%).

The system buffer and line buffer can be directly manipulated using the following functions.

(**CLEARBUF** *FILE FLG*) [Function]

> Clears the input buffer for *FILE*. If *FILE* is T and *FLG* is T, the contents of Interlisp's system buffer and line buffer are saved (and can be obtained via SYSBUF and LINBUF described below).

> When you type Control-D or Control-E, or any of the interrupt characters that require terminal interaction (Control-G, or Control-P), Interlisp automatically performs (CLEARBUF T T). For Control-P and, when the break is exited normally, control-H, Interlisp restores the buffer after the interaction.

> The action of (CLEARBUF T), i.e., clearing of typeahead, is also available as the RUBOUT interrupt character, initially assigned to the delete key in Interlisp-D. Note that this interrupt clears both buffers at the time it is typed, whereas the action of the CHARDELETE and LINEDELETE character occur at the time they are read.

(**SYSBUF** *FLG*) [Function]

> If *FLG* = T, returns the contents of the system buffer (as a string) that was saved at the last (CLEARBUF T T). If *FLG* = NIL, clears this internal buffer.

(**LINBUF** *FLG*) [Function]

> Same as SYSBUF for the line buffer.

If both the system buffer and Interlisp's line buffer are empty, the internal buffers associated with LINBUF and SYSBUF are not changed by a (CLEARBUF T T).

(**BKSYSBUF** *X FLG RDTBL*) [Function]

> BKSYSBUF appends the PRIN1-name of *X* to the system buffer. The effect is the same as though the user typed *X*. Some implementations have a limit on the length of *X*, in which case characters in *X* beyond the limit are ignored. Returns *X*.

> If *FLG* is T, then the PRIN2-name of *X* is used, computed with respect to the readtable *RDTBL*. If *RDTBL* is NIL or omitted, the current readtable of the TTY process (which is to receive the characters) is used. Use this for copy selection functions that want their output to be a readable expression in an Exec.

> Note that if you are typing at the same time as the BKSYSBUF is being performed, the relative order of the typein and the characters of *X* is unpredictable.

(**BKLINBUF** *STR*) [Function]

STR is a string. BKLINBUF sets Interlisp's line buffer to *STR*. Some implementations have a limit on the length of *STR*, in which case characters in *STR* beyond the limit are ignored. Returns *STR*.

(**BKSYSCHARCODE** *CODE*)   [Function]

This function appends the character code *CODE* to the system input buffer. The function BKSYSBUF is implemented by repeated calls to BKSYSCHARCODE.

BKLINBUF, BKSYSBUF, LINBUF, and SYSBUF provide a way of "undoing" a CLEARBUF. Thus to "peek" at various characters in the buffer, one could perform (CLEARBUF T T), examine the buffers via LINBUF and SYSBUF, and then put them back.

The more common use of these functions is in saving and restoring typeahead when a program requires some unanticipated (from the user's standpoint) input. The function RESETBUFS provides a convenient way of simply clearing the input buffer, performing an interaction with the user, and then restoring the input buffer.

(**RESETBUFS** $FORM_1$, $FORM_2$,... *FORMN*)   [NLambda NoSpread Function]

Clears any typeahead (ringing the terminal's bell if there was, indeed, typeahead), evaluates $FORM_1$, $FORM_2$,... $FORM_N$, then restores the typeahead. Returns the value of $FORM_N$. Compiles open.

Dribble Files

A dribble file is a "transcript" of all of the input and output on a terminal. In Interlisp-D, DRIBBLE opens a dribble file for the current process, recording the terminal input and output for that process. Multiple processes can have separate dribble files open at the same time.

(**DRIBBLE** *FILE APPENDFLG THAWEDFLG*)   [Function]

Opens *FILE* and begins recording the typescript. Returns the old dribble file if any, otherwise NIL. If *APPENDFLG* = T, the typescript will be appended to the end of *FILE*. If *THAWEDFLG* = T, the file will be opened in "thawed" mode, for those implementations that support it. (DRIBBLE) closes the dribble file for the current process. Only one dribble file can be active for each process at any one time, so (DRIBBLE FILE1) followed by (DRIBBLE FILE2) will cause FILE1 to be closed.

(**DRIBBLEFILE**)   [Function]

Returns the name of the current dribble file for the current process, if any, otherwise NIL.

Terminal input is echoed to the dribble file a line buffer at a time. Thus, the typescript produced is somewhat neater than that appearing on the user's terminal, because it does not show characters that were erased via Control-A or Control-Q. Note that the typescript file is not included in the list of files returned by (OPENP), nor will it be closed by a call to CLOSEALL or CLOSEF. Only (DRIBBLE) closes the typescript file.

Cursor and Mouse

A mouse is a small box connected to the computer keyboard by a long wire. On the top of the mouse are two or three buttons. On the bottom is a rolling ball or a set of photoreceptors, to detect when the mouse is moved. As the mouse is moved on a surface, a small image on the screen, called the cursor, moves to follow the movement of the mouse. By moving the mouse, the user can cause the cursor to point to any part of the display screen.

The mouse and cursor are an important part of the Interlisp-D user interface. The Interlisp-D window system allows the user to create, move, and reshape windows, and to select items from displayed menus, all by moving the mouse and clicking the mouse buttons. This section describes the low-level functions used to control the mouse and cursor.

Changing the Cursor Image

Interlisp-D maintains the image of the cursor on the screen, moving it as the mouse is moved. The bitmap that becomes visible as the cursor can be accessed by the following function:

(**CURSORBITMAP**)   [Function]

> Returns the cursor bitmap.

CURSORWIDTH   [Variable]
CURSORHEIGHT   [Variable]

> Value is the width and height of the cursor bitmap, respectively.

The cursor bitmap can be changed like any other bitmap by BITBLTing into it or pointing a display stream at it and printing or drawing curves. The CURSOR datatype has the following field names CUBITSPERPIXEL CUIMAGE, CUMASK, CUHOTSPOTX, CUHOTSPOTY, CUDATA

CURSOR objects can be saved on a file using the file package command CURSORS, or the UGLYVARS file package command.

(**CURSORCREATE** *BITMAP X Y*)   [Function]

> Returns a cursor object which has *BITMAP* as its image and the location (*X,Y*) as the hot spot. If *X* is a POSITION, it is used as the hot spot. If *BITMAP* has dimensions different from CURSORWIDTH by CURSORHEIGHT, the lesser of the widths and the lesser of the

heights are used to determine the bits that actually get copied into the lower left corner of the cursor. If *X* is NIL, **0** is used. If *Y* is NIL, CURSORHEIGHT-1 is used. The default cursor is an uparrow with its tip in the upper left corner and its hot spot at (0,CURSORHEIGHT-1).

(**CURSOR** *NEWCURSOR* −)   [Function]

Returns a CURSOR record instance that contains (a copy of) the current cursor specification. If *NEWCURSOR* is a CURSOR record instance, the cursor will be set to the values in *NEWCURSOR*. If *NEWCURSOR* is T, the cursor will be set to the default cursor

DEFAULTCURSOR, an upward left pointing arrow: .

(**SETCURSOR** *NEWCURSOR* −)   [Function]

If *NEWCURSOR* is a CURSOR record instance, the cursor will be set to the values in *NEWCURSOR*. This does not return the old cursor, and therefore, provides a way of changing the cursor without using storage.

(**FLIPCURSOR**)   [Function]

Inverts the cursor.

The following list describes the cursors used by the Interlisp-D system. Most of them are stored as the values of various variables.

 In variable DEFAULTCURSOR. This is the default cursor.

 In variable WAITINGCURSOR. Represents an hourglass. Used during long computations.

 In variable MOUSECONFIRMCURSOR. Indicates that the system is waiting for the user to confirm an action by pressing the left mouse button, or aborting the action by pressing any other button. Used by the function MOUSECONFIRM (see Chapter 28).

 In variable SYSOUTCURSOR. Indicates that the system is saving the virtual memory in a sysout file. See SYSOUT, Chapter 12.

 In variable SAVINGCURSOR. Indicates that SAVEVM has been called automatically to save the virtual memory state after the system is idle for long enough. See SAVEVMWAIT, Chapter 12.

 In variable CROSSHAIRS. Used by GETPOSITION (see Chapter 28) to indicate a position.

In variable `BOXCURSOR`.   Used by `GETBOXPOSITION` (see Chapter 28) to indicate where to place the corner of a box.

In variable `FORCEPS`.   Used by `GETREGION` (see Chapter 28) when the user switches corners.

In variable `EXPANDINGBOX`.   Used by `GETREGION` (see Chapter 28) when a box is first displayed.

In variable `UpperRightCursor`.

In variable `LowerRightCursor`.

In variable `UpperLeftCursor`.

In variable `LowerLeftCursor`.

The previous four cursors are used by `GETREGION` (see Chapter 28) to indicate the four corners of a region.

In variable `VertThumbCursor`. Used during scrolling to indicate thumbing in a vertical scroll bar.

In variable `VertScrollCursor`.

In variable `ScrollUpCursor`.

In variable `ScrollDownCursor`.

The previous four cursors are used by `SCROLL.HANDLER` (see Chapter 28) during vertical scrolling.

In variable `HorizThumbCursor`. Used during scrolling to indicate thumbing in a horizontal scroll bar.

In variable `HorizScrollCursor`.

In variable `ScrollLeftCursor`.

In variable `ScrollRightCursor`.

The previous four cursors are used by `SCROLL.HANDLER` (see Chapter 28) during horizontal scrolling.

These cursors are used by the Teleraid low-level debugger. These cursors are not accessable as standard Interlisp-D cursors.

Flashing Bars on the Cursor

The low-level Interlisp-D system uses the cursor to display certain system status information, such as garbage collection or swapping. This is done because changing the cursor image is very quick, and does not require interacting with the window system. Interlisp inverts horizontal bars on the cursor when the system is swapping pages, or doing certain stack operations. Normally, these bars are only inverted for a very short time, so they look like they are flashing. These cursor changes are interpreted as follows:

Inverted cursor: Whatever image is being displayed as the cursor, whenever Interlisp does a garbage collection, the whole cursor is inverted.

Top bar: Swap read. On when Interlisp is swapping in a page from the virtual memory file into the real memory. It is also on when Interlisp allocates a new virtual memory page, even though that doesn't involve a disk read. If this is flashing a lot, the system is doing a lot of swapping. This is an indication that the virtual memory working set is fragmented (see Chapter 22). Performance may be improved by reloading a clean Interlisp system.

Upper middle bar: Stack operations. If this is flashing a lot, it suggests that some process is neglecting to release stack pointers in a timely fashion (see Chapter 11).

Lowereler middle bar: Stack operations. On when Interlisp is moving frames on the stack. If the system is slow, and this is flashing a lot, HARDRESET (see Chapter 23) sometimes helps.

Bottom bar: Swap write. On when Interlisp writes a dirty virtual memory page from the real memory back into the virtual memory file.

Cursor Position

The position at which the cursor bitmap is being displayed can be read or set using the following functions:

(**CURSORPOSITION** *NEWPOSITION DISPLAYSTREAM OLDPOSITION*)　　　　　[Function]

Returns the location of the cursor in the coordinate system of *DISPLAYSTREAM* (or the current display stream, if *DISPLAYSTREAM* is NIL). If *NEWPOSITION* is non-NIL, it

should be a position and the cursor will be positioned at *NEWPOSITION*. If *NEWPOSITION* is NIL, the current position is simple returned.

The current position of the cursor is the position of the "hot spot" of the cursor, not the position of the cursor bitmap.

If *OLDPOSITION* is a POSITION object, this object will be changed to point to the location of the cursor and returned, rather of allocating a new POSITION. This can improve performance if CURSORPOSITION is called repeatedly to track the cursor.

To get the location of the cursor in absolute screen coordinates, use the variables LASTMOUSEX and LASTMOUSEY.

(**ADJUSTCURSORPOSITION** *DELTAX DELTAY*)   [Function]

Moves the cursor *DELTAX* points in the X direction and *DELTAY* points in the Y direction. *DELTAX* and *DELTAY* default to 0.

Mouse Button Testing

There are two or three keys on the mouse. These keys (also called buttons) are referred to by their location: LEFT, MIDDLE, or RIGHT. The following macros are provided to test the state of the mouse buttons:

(**MOUSESTATE** *BUTTONFORM*)   [Macro]

Reads the state of the mouse buttons, and returns T if that state is described by *BUTTONFORM*. *BUTTONFORM* can be one of the key indicators LEFT, MIDDLE, or RIGHT; the atom UP (indicating all keys are up); the form (ONLY KEY); or a form of AND, OR, or NOT applied to any valid button form.

For example: (MOUSESTATE LEFT) will be true if the left mouse button is down. (MOUSESTATE (ONLY LEFT)) will be true if the left mouse button is the only one down. (MOUSESTATE (OR (NOT LEFT) MIDDLE)) will be true if either the left mouse button is up or the middle mouse button is down.

(**LASTMOUSESTATE** *BUTTONFORM*)   [Macro]

Similar to MOUSESTATE, but tests the value of LASTMOUSEBUTTONS (below) rather than getting the current state. This is useful for determining which keys caused MOUSESTATE to be true.

(**UNTILMOUSESTATE** *BUTTONFORM INTERVAL*)   [Macro]

BUTTONFORM is as described in MOUSESTATE. Waits until *BUTTONFORM* is true or until *INTERVAL* milliseconds have elapsed. The value of UNTILMOUSESTATE is T if *BUTTONFORM* was satisfied before it timed out, otherwise NIL. If *INTERVAL* is NIL, it waits indefinitely. This compiles into an open loop that calls the TTY wait background

function. This form should not be used inside the `TTY` wait background function. `UNTILMOUSESTATE` does not use any storage during its wait loop.

Low Level Mouse Functions

This section describes the functions and variables that provide low level access to the mouse and cursor.

(**LASTMOUSEX** *DISPLAYSTREAM*)   [Function]

> Returns the value of the cursor's X position in the coordinates of *DISPLAYSTREAM* (as of the last call to GETMOUSESTATE, below).

(**LASTMOUSEY** *DISPLAYSTREAM*)   [Function]

> Returns the value of the cursor's Y position in the coordinates of *DISPLAYSTREAM* (as of the last call to GETMOUSESTATE, below).

LASTMOUSEX   [Variable]

> Value is the X position of the cursor in absolute screen coordinates (as of the last call to GETMOUSESTATE, below).

LASTMOUSEY   [Variable]

> Value is the Y position of the cursor in absolute screen coordinates (as of the last call to GETMOUSESTATE, below).

LASTMOUSEBUTTONS   [Variable]

> Value is an integer that has bits on corresponding to the mouse buttons that are down (as of the last call to GETMOUSESTATE, below). Bit 4Q is the left mouse button, 2Q is the right button, 1Q is the middle button.

LASTKEYBOARD   [Variable]

> Value is an integer encoding the state of certain keys on the keyboard (as of the last call to GETMOUSESTATE, below). Bit 200Q = lock, 100Q = left shift, 40Q = ctrl, 10Q = right shift, 4Q = blank Bottom, 2Q = blank Middle, 1Q = blank Top. If the key is down, the corresponding bit is on.

(**GETMOUSESTATE**)   [Function]

> Reads the current state of the mouse and sets the variables LASTMOUSEX, LASTMOUSEY, and LASTMOUSEBUTTONS. In polling mode, the program must remember the previous state and look for changes, such as a key going up or down, or the cursor moving outside a region of interest.

(**DECODEBUTTONS** *BUTTONSTATE*)   [Function]

Returns a list of the mouse buttons that are down in the state *BUTTONSTATE*. If *BUTTONSTATE* is not a small integer, the value of LASTMOUSEBUTTONS (above) is used. The button names that can be returned are: LEFT, MIDDLE, RIGHT (the three mouse keys).

Keyboard Interpretation

For each key on the keyboard and mouse there is a corresponding bit in memory that the hardware turns on and off as the key moves up and down. System-level routines decode the meaning of key transitions according to a table of "key actions", which may be to put particular character codes in the sysbuffer, cause interrupts, change the internal shift/control status, or create events to be placed in the mouse buffer.

(**KEYDOWNP** *KEYNAME*)   [Function]

Used to read the instantaneous state of any key, independent of any buffering or pre-assigned key action. Returns T if the key named *KEYNAME* is down at the moment the function is executed.

Most keys are named by the characters on the key-top. Therefore, (KEYDOWNP 'a) or (KEYDOWNP 'A) returns T if the "A" key is down.

There are a number of keys that do not have standard names printed on them. These can be accessed by special names as follows:

Space	SPACE
Carriage return	CR
Line-feed	LF
Backspace	BS
Tab	TAB
Blank keys on 1132	The 1132 keyboard has three unmarked keys on the right of the normal keyboard. These can be accessed by BLANK-BOTTOM, BLANK-MIDDLE, and BLANK-TOP.
Escape	ESCAPE
Shift keys	LSHIFT for the left shift key, RSHIFT for the right shift key.
Shift lock key	LOCK
Control key	CTRL
Mouse buttons	The state of the mouse buttons can be accessed using LEFT, MIDDLE, and RIGHT.

If `KEYNAME` is a small integer, it is taken to be the internal key number.   Otherwise, it is taken to be the name of the key. This means, for example, that the name of the "6" key is not the number 6. Instead, spelled-out names for all the digit keys have been assigned. The "6" key is named `SIX`. It happens that the key number of the "6" key is 2. Therefore, the following two forms are equivalent:

> (KEYDOWNP 'SIX)
>
> (KEYDOWNP 2)

(**SHIFTDOWNP** *SHIFT*)   [Function]

Returns `T` if the internal "shift" flag specified by *SHIFT* is on; `NIL` otherwise.

If *SHIFT* = `1SHIFT`, `2SHIFT`, `LOCK`, `META`, or `CTRL`, SHIFTDOWNP returns the state of the left shift, right shift, shift lock, control, and meta flags, respectively.

If *SHIFT* = `SHIFT`, SHIFTDOWNP returns `T` if either the left or right shift flag is on.

If *SHIFT* = `USERMODE1`, `USERMODE2`, or `USERMODE3`, SHIFTDOWNP returns the state of one of three user-settable flags that have no other effect on key interpretation. These flags can be set or cleared on character transitions by using `KEYACTION` (below).

(**KEYACTION** *KEYNAME ACTIONS* −)   [Function]

Changes the internal tables that define the action to be taken when a key transition is detected by the system keyboard handler. *KEYNAME* is specified as for KEYDOWNP. *ACTIONS* is a dotted pair of the form (`DOWN-ACTION` . `UP-ACTION`), where the acceptable transition actions and their interpretations are:

> NIL

> IGNORE   Take no action on this transition (the default for up-transitions on all ordinary characters).

(CHAR SHIFTEDCHAR LOCKFLAG)

> > If a transition action is a three-element list, `CHAR` and `SHIFTEDCHAR` are either character codes or (non-numeric) single-character litatoms standing for their character codes.   Note that `CHAR` and `SHIFTEDCHAR` can be full sixteen-bit NS characters (see page `X.XX`). When the transition occurs, `CHAR` or `SHIFTEDCHAR` is transmitted to the system buffer, depending on whether either of the two shift keys are down.

> > `LOCKFLAG` is optional, and may be `LOCKSHIFT` or `NOLOCKSHIFT`. If `LOCKFLAG` is `LOCKSHIFT`, then `SHIFTEDCHAR` will also be transmitted when the `LOCK` shift is down (the alphabetic keys initially specify `LOCKSHIFT`, but the digit keys specify `NOLOCKSHIFT`). For

example, `(a A LOCKSHIFT)` and `(61Q ! NOLOCKSHIFT)` are the initial settings for the down transitions of the "a" and "1" keys respectively.

`1SHIFTUP, 1SHIFTDOWN`

`2SHIFTUP, 2SHIFTDOWN`

`CTRLUP, CTRLDOWN`

`METAUP, METADOWN`   Change the status of the internal "shift" flags for the left shift, right shift, control, and meta keys, respectively. These shifts affect the interpretation of ordinary key actions. If either of the shifts is down, then `SHIFTEDCHAR`s are transmitted. If the control flag is on, then the the seventh bit of the character code is cleared as characters are transmitted. If the meta flag is on, the the eighth bit of the character code is set (normally cleared) as characters are transmitted. For example, the initial keyactions for the left shift key is (`1SHIFTDOWN . 1SHIFTUP`).

`LOCKUP, LOCKDOWN, LOCKTOGGLE`

Change the status of the internal "shift" flags for the shift lock key. If the lock flag is down, then `SHIFTEDCHAR`s are transmitted if the key action specified `LOCKSHIFT`. `LOCKUP` and `LOCKDOWN` clear and set the shift lock flag, respectively. `LOCKTOGGLE` complements the flag (turning it off if the flag is on; on if the flag is off).

`USERMODE1UP, USERMODE1DOWN, USERMODE1TOGGLE`

`USERMODE2UP, USERMODE2DOWN, USERMODE2TOGGLE`

`USERMODE3UP, USERMODE3DOWN, USERMODE3TOGGLE`

Change the status of the three user flags `USERMODE1`, `USERMODE2`, and `USERMODE3`, whose status can be determined by calling `SHIFTDOWNP` (above). These flags have no other effect on key interpretation.

`EVENT`   An encoding of the current state of the mouse and selected keys is placed in the mouse-event buffer when this transition is detected.

`KEYACTION` returns the previous setting for `KEYNAME`. If `ACTIONS` is `NIL`, returns the previous setting without changing the tables.

(**MODIFY.KEYACTIONS** *KEYACTIONS SAVECURRENT?*)   [Function]

KEYACTIONS is a list of key actions to be set, each of the form (KEYNAME . ACTIONS). The effect of `MODIFY.KEYACTIONS` is as if (`KEYACTION KEYNAME ACTIONS`) were performed for each item on *KEYACTIONS*.

If *SAVECURRENT?* is non-NIL, then MODIFY.KEYACTIONS returns a list of all the results from *KEYACTION*, otherwise it returns NIL. This can be used with a MODIFY.KEYACTIONS that appears in a RESETFORM, so that the list is built at "entry", but not upon "exit".

(**METASHIFT** *FLG*) [NoSpread Function]

If *FLG* is T, changes the keyboard handler (via KEYACTION) so as to interpret the "stop" key on the 1108 as a metashift: if a key is struck while the meta is down, it is read with the 200Q bit set. For CHAT users this is a way of getting an "Edit" key on your simulated Datamedia.

If *FLG* is other than NIL or T, it is passed as the ACTIONS argument to KEYACTION. The reason for this is that if someone has set the "STOP" key to some random behavior, then (RESETFORM (METASHIFT T) --) will correctly restore that random behavior.

Display Screen

Medley supports a high-resolution bitmap display screen. All printing and drawing operations to the screen are actually performed on a bitmap in memory, which is read by the computer hardware to become visible as the screen. This section describes the functions used to control the appearance of the display screen.

(**SCREENBITMAP**) [Function]

Returns the screen bitmap.

SCREENWIDTH [Variable]
SCREENHEIGHT [Variable]

Value is the width and height of the screen bitmap, respectively.

WHOLEDISPLAY [Variable]

Value is a region that is the size of the screen bitmap.

The background shade of the display window can be changed using the following function:

(**CHANGEBACKGROUND** *SHADE* —) [Function]

Changes the background shade of the window system. *SHADE* determines the pattern of the background. If *SHADE* is a texture, then the background is simply painted with it. If *SHADE* is a BITMAP, the background is tesselated (tiled) with it to cover the screen. If *SHADE* is T, it changes to the original shade, the value of WINDOWBACKGROUNDSHADE. It returns the previous value of the background.

WINDOWBACKGROUNDSHADE [Variable]

Value is the default background shade for the display.

(**VIDEOCOLOR** *BLACKFLG*)   [NoSpread Function]

Sets the interpretation of the bits in the screen bitmap. If *BLACKFLG* is NIL, a 0 bit will be displayed as white, otherwise a 0 bit will be displayed as black. VIDEOCOLOR returns the previous setting. If *BLACKFLG* is not given, VIDEOCOLOR will return the current setting without changing anything.

Note: This function only works on the Xerox 1100 and Xerox 1108.

(**VIDEORATE** *TYPE*)   [Function]

Sets the rate at which the screen is refreshed. *TYPE* is one of NORMAL or TAPE. If *TYPE* is TAPE, the screen will be refreshed at the same rate as TV (60 cycles per second). This makes the picture look better when video taping the screen. Note: Changing the rate may change the dimensions of the display on the picture tube.

Maintaining the video image on the screen uses cpu cycles, so turning off the display can improve the speed of compute-bound tasks. When the display is off, the screen will be white but any printing or displaying that the program does will be visible when the display is turned back on.

Note: Breaks and PAGEFULLFN waiting (see Chapter 28) turn the display on, but users should be aware that it is possible to have the system waiting for a response to a question printed or a menu displayed on a non-visible part of the screen. The functions below are provided to turn the display off.

Note: These functions have no effect on the Xerox 1108 display.

(**SETDISPLAYHEIGHT** *NSCANLINES*)   [Function]

Sets the display to only show the top *NSCANLINES* of the screen. If *NSCANLINES* is T, resets the display to show the full screen. Returns the previous setting.

(**DISPLAYDOWN** *FORM NSCANLINES*)   [Function]

Evaluates *FORM* (with the display set to only show the top *NSCANLINES* of the screen), and returns the value of *FORM*. It restores the screen to its previous setting. If *NSCANLINES* is not given, it defaults to 0.

Miscellaneous Terminal I/O

(**RINGBELLS** *N*)   [Function]

Flashes (reverse-videos) the screen *N* times (default 1). On the Xerox 1108, this also beeps through the keyboard speaker.

(**PLAYTUNE** *Frequency/Duration.pairlist*)   [Function]

On the Xerox 1108, PLAYTUNE plays a sequence of notes through the keyboard speaker. *Frequency/Duration.pairlist* should be a list of dotted pairs (*FREQUENCY . DURATION*). PLAYTUNE maps down its argument, beeping the 1108 keyboard buzzer at each frequency for the specified amount of time. Specifying NIL for a frequency means to turn the beeper off the specified amount of time. The units of time are TICKS (Chapter 12), which last about 28.78 microseconds on the Xerox 1108. PLAYTUNE makes no sound on a Xerox 1132. The default "simulate" entry for Control-G (ASCII BEL) on the 1108 uses PLAYTUNE to make a short beep.

PLAYTUNE is implemented using BEEPON and BEEPOFF:

(**BEEPON** *FREQ*) [Function]

On the Xerox 1108, turns on the keyboard speaker playing a note with frequency *FREQ*, measured in Hertz (see Chapter 12). The speaker will continue to play the note until BEEPOFF is called.

(**BEEPOFF**) [Function]

Turns off the keyboard speaker on the Xerox 1108.

(**SETMAINTPANEL** *N*) [Function]

On the Xerox 1108, this sets the four-digit "maintanance panel" display on the front of the computer to display the number *N*.