INTERLISP

The Language and Its Usage

STEPHEN H. KAISLER
Defense Advanced Research Projects Agency

A Wiley-Interscience Publication
JOHN WILEY & SONS
New York • Chichester • Brisbane • Toronto • Singapore

Library of Congress Cataloging in Publication Data:

Kaisler, Stephen H. (Stephen Hendrick)
INTERLISP: the language and its usage.

"A Wiley-Interscience publication."
Bibliography: p.
Includes index.
1. LISP (Computer program language) I. Title
QA76.73.L23K35 1986 005.13'3 85-31528
ISBN 0-471-81644-2

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To my loving wife, Chryl Kennedy Kaisler,

who has persevered through the many hours that it took to write this book. Her patience and support are greatly appreciated and, I hope, amply rewarded.

Preface

This text describes the features of a dialect of LISP known as INTERLISP. INTERLISP stands for "Interactive Lisp." It provides a rich program development and problem prototyping environment.

There is strong agreement among LISP developers and users about what the basic functions of LISP are. All of these are described in the first few chapters of this book. But there is no "standard" for LISP and it is unlikely that we will see one in the near future. To date, the best standard that exists is the INTERLISP Reference Manual. This manual is edited and maintained by staff members of the Xerox Corporation's Palo Alto Research Center, Palo Alto, CA and Special Information Systems Division, Pasadena, CA.

For all of the detail and wealth of information contained in the INTERLISP Reference Manual [irm78, irm83], it remains a remarkably obtuse document. Some say that it is user-hostile. One must have extreme familiarity with INTERLISP, usually through the tutelage of an INTERLISP guru, to be able to use it efficiently. This text attempts to correct this problem by describing most of the major functions, capabilities, and packages provided by INTERLISP. It is augmented by numerous examples taken from actual experience as well as many technical papers published in the open AI literature. Where appropriate, references to the INTERLISP Reference Manual [irm78, irm83] will be made using the notation (IRM x.y.z) where x,y,z refer to the chapter, section, and paragraph of the IRM respectively.

This text is based upon experience gained using INTERLISP on an IBM 3081 under VM/SP (version 3.0), INTERLISP-D on a Xerox 1100 Scientific Information Processor (through the Fugue release), and INTERLISP-10 on a DECSystem-20 under TOPS-20 (release dated 26-SEP-83).

It is not my intent to teach you how to "program" in LISP in this text. There are numerous books that explain the essential features of LISP programming from the viewpoint of the novice. Many of these are mentioned in the references, but a notable volume is that of Winston and Horn:

Winston, P. and Horn, B.K.P.
Lisp
Addison-Wesley, Reading, MA, 1981

whose first few chapters provide a basic introduction to LISP programming. If you have not programmed in LISP before, you may also want to consult

Touretzsky, D.
LISP: A Gentle Introduction to Symbolic Computation
Harper & Row, New York, 1983

Wilensky, R.
LISPCraft
W. Norton & Sons, San Francisco, 1984

The text is divided at a logical breakpoint. Chapters 1 through 16 discuss features that are largely found in most LISP systems, although they may often go by different names or have different implementations. Chapters 17 through 31 discuss features that provide INTERLISP with its power as an interactive programming environment. A second volume is contemplated that will discuss the INTERLISP-D implementation. Tentatively, this volume will be entitled

Interlisp: The Interactive Programming Environment

You will find this current text replete with examples of LISP functions that have been described and (sometimes) commented. This reflects a bias of mine that you understand a language by reading programs or functions written in the language. That is, we all learn programming by analogy; we look at how somebody else has written a program and copy the essential elements. Sometimes we copy their style and faults as well. The functional code presented in this text has, for the most part, been tested on one of the INTERLISP systems mentioned above.

STEPHEN H. KAISLER

Silver Spring, Maryland
March, 1986

Acknowledgments

No book is ever completed without the assistance of a great many people. Let me take this opportunity to thank a few of those who assisted me. Jerry Alexander of Analytics, Inc. encouraged the completion of the book as did numerous Interlisp programmers and adherents who believed that something better than the IRM was needed. Charles Kellogg of MCC and John Vittal and Beau Shiel of Xerox reviewed the manuscript and provided many cogent comments. Jim Gaughan, my former editor; Carol Beasley, who stepped in in his absence; and Maria Taylor, my current editor, were all instrumental in seeing the project through to completion. Also, I would like to especially thank Sandra Renner, their secretary, who handled numerous queries and phone calls. Finally, I want to acknowledge the contribution of the family cats, Sprite and Gizmo, who carefully monitored the progress of the book by sitting in front of the computer terminal while I typed away.

S.H.K.

Contents

INTERLISP

1

Introduction

LISP, as a language, has been around for about 25 years [mcca78]. It was originally developed to support artificial intelligence (AI) research. At first, it seemed to be little noticed except by a small band of academics who implemented some of the early LISP interpreters and wrote some of the early AI programs. In the early 60's, LISP began to diverge as various implementations were developed for different machines. McCarthy [mcca78] gives a short history of its early days.

LISP, as a programming language, began to be widely used in the early 70's. A number of organizations supported different dialects—BBN, MIT, Xerox PARC, UC Irvine, and others. In the middle 70's, several people realized that conventional machines were not suitable for the efficient execution of LISP programs. They began to develop a class of specialized processors known as LISP machines. By the early 80's, renewed interest in AI and expert systems caused LISP to become more visible. Several dialects had been more or less standardized and a few companies (such as Xerox, Symbolics, and Lisp Machine) had entered into commercial production of LISP machines. Moreover, many people had discovered that LISP provided an excellent implementation language for some of the new ideas in software engineering.

Today, LISP is being used for many applications other than AI programs (for example, see [elli80] and [levi80]), although it is still strongly associated with that discipline. In order to further its acceptance as a general purpose programming language, I have focused on describing a specific dialect of LISP—INTERLISP. I chose INTERLISP because there is a well-defined standard for it—namely, the INTERLISP Reference Manual [irm78, irm83]. Moreover, INTERLISP is available on the DECSystem 10/20 under TENEX and TOPS-20, on the VAX-11/780 under UNIX and VMS, on the IBM 30xx series under VM/SP (available from Uppsala University, Sweden), and on the Xerox 1100 family of scientific information processors. Undoubtedly, more implementations will be available in the near future.

This chapter will set the stage for an in-depth analysis of INTERLISP. In the first two sections, we motivate the choice of LISP and briefly explore some of

the available dialects. Sections 1.3 and 1.4 discuss the impact of LISP on the way we program and on the way we think about the architecture of software systems. Section 1.5 reviews the structure of the text. Section 1.6 discusses presentation issues.

1.1 WHY LISP?

Why should you choose LISP to implement a software system? For many reasons, as it turns out, some of which are physical (e.g., implementation issues), some of which are stylistic (e.g., programming issues), and some of which are conceptual (e.g., design or architectural issues). For the moment, let us concentrate upon the physical reasons.

LISP offers many features and capabilities that would be/are difficult to provide in more traditional languages such as FORTRAN, PASCAL, and PL/1. I would like to review these with you and argue for the view that LISP can do them better. This does not imply that LISP is the best language for all applications. There is no such language. But I do take the view that LISP should have a well-defined niche in your repertoire of programming languages.

With this view in mind, the following paragraphs discuss some of the key ideas associated with LISP and contrast their implementation in conventional languages.

1.1.1 Symbolic Computation

Most programs compute on numbers. Conventional languages make the task of manipulating numbers quite easy. They may also be able to handle string or character expressions in an efficient manner. But, as AI research has shown, much computation at the human level is done on concepts—complex, structured representations of knowledge. Representation of knowledge is often the most critical element of the computational problem. There may be no intuitively obvious way of selecting numeric codes to represent pieces of knowledge. Arbitrary encodings tend to obscure significant aspects of the problem, and make the task of programming, debugging, and testing very complicated.

LISP was designed to allow the efficient, easy representation of symbolic expressions. We use lists in everyday life to keep track of things, so our personal experience should readily transfer to our use of lists in the computer. LISP essentially recognizes two basic data structures: atoms and lists. Atoms are just symbols—indivisible sequences of characters that have a meaning and a value. Lists are collections of atoms and/or lists (e.g., sublists). Lists are usually stored in the computer in a form similar to the way we see them on paper. The sequencing information of the elements of a list is explicitly stored with each list item.

To manipulate symbolic knowledge, we represent it as a list of atoms and then perform various operations upon the representation. In many cases, the manipulation functions modify the structure of the list to add or subtract infor-

mation, to alter its structure, or to transform its contents. Examples throughout the text will demonstrate how these operations are performed.

A number of languages have been built "on top of" LISP to provide symbolic algebraic manipulation, most notably, REDUCE2, developed by A.C. Hearn and colleagues at the University of Utah, and MACSYMA, developed by the MATHLAB Group at the Laboratory for Computer Science, MIT [math77]. Nicol [nico81] provides a definition of a simple symbolic differentiator for algebraic expressions.

1.1.2 Information Representation by Lists

Information representation is a critical problem in designing a software system. It is particularly acute in AI programs and expert systems because the information structures themselves are manipulated. In LISP, most information is represented as lists. Simple lists are composed of atoms while complex expressions have multiple levels of lists. This capability allows us to build very complex information structures without too much regard for addressing. By comparison, conventional languages rarely provide more than linear data structures, including two dimensional arrays and record structures as in PASCAL.

Consider the representation of information concerning familial relationships between individuals. To represent the fact that Isaac is a child of Abraham, we may use the list

```
(is-child-of isaac abraham)
```

and the corresponding list to represent the relationship that Abraham is a parent of Isaac

```
(is-parent-of abraham isaac).
```

Other facts that might be represented about individuals include physical data such as gender, age, height, and so on, and other relationships, skills, or occupations. The power of the list representation allows us to capture all of this information in a single uniform data structure, the list. With a few simple primitive functions, we may begin to access and manipulate this information, and even build more powerful operators.

1.1.3 Primitive Functions

The essential kernel of LISP [moor79] is composed of a small set of primitive functions. Usually, these functions are hardwired to improve the efficiency of their execution. The composition of the kernel depends on the dialect. For example, the kernel of INTERLISP/370 is written in IBM assembly language while the kernel of INTERLISP-D is written in microcode.

Most conventional languages do not have a primitive set of functions. Rather, the primitive functions reside in the underlying machine language. Conventional languages such as FORTRAN and PASCAL consist of statement types which represent simple methods for writing (possibly long) sequences of primitive functions. The source statements are translated into a sequence of machine language instructions by a compiler or an interpreter. The resulting executable program often bears no resemblance to the source program.

John Backus [back78] argues for a simpler style of programming based on the concept of "functional programming." It is interesting to note that functional programming corresponds to the basic mathematical theory of function composition carried over to the realm of programming language development and methodology.

LISP provides an excellent example of the notions expressed in Backus's paper. The minimal set of functions that we need in LISP to implement all other functions are CAR, CDR, and CONS, EQ, and a few predicates for testing data types. All other functions may be built from compositions of these elementary or primitive functions. Of course, most LISP systems define a much larger set of primitive functions for reasons of efficiency. Many of the functions that are described in this text for which definitions are given exemplify the notion of function composition.

1.1.4 Function Composition

Proceeding from the kernel, LISP users create more complex functions by combining expressions containing primitive functions. The procedure is directly analogous to mathematical function composition. The beauty of this feature is that most of INTERLISP is written in LISP. One merely keeps building functions on top of functions to create more complex systems.

We should think of LISP as a tool for building more powerful tools. Many systems designed for implementing artificial intelligence applications may rightfully be considered extended programming languages. Good examples include OPS5 [forg81] developed by Charles Forgy at CMU and ROSIE [fain81, 82, haye82] developed by Frederick Hayes-Roth and others at the Rand Corporation.

1.1.5 Functions versus Data

LISP possesses a singular advantage over most conventional languages, even those that are interpreted. This is that functions and data are represented in the same way. Conventional languages stress the separation of code and data as independent entities. LISP emphasizes uniformity of representation—a single model, if you will. Thus, we can create a data structure as a list and treat the same data structure as an executable function definition or function invocation.

The beauty of this approach is demonstrated in a technique known as procedural attachment [stee79]. You may define an object, nominally a frame, which

has slots for the attributes that describe or distinguish the object from others of its type. The value of an attribute may be an integer, atom, string, pointer to another object, or a procedure (e.g., a LISP function definition). The procedure may do anything a LISP function can do since it is a LISP function. Common uses of procedures are validation of data, transformation of data, asking questions to obtain the values of other attributes, or performing some computation to instantiate another instance of the frame. You may operate upon the procedure definition as data, but execute it by applying EVAL to it.

1.1.6 The EVAL Function

The function EVAL serves a dual purpose in LISP. It is both the formal definition of the language as well as an interpreter for its execution. That is, EVAL validates the syntax of the expressions presented to it for execution and verifies their semantics by invoking the requested functions. By comparison, most conventional languages separate the syntax validation, performed by the compiler, from the semantic verification, performed by the run-time monitor.

1.2 LISP DIALECTS

LISP has a long and rich history. There have been many dialects of LISP implemented on many different machines. McCarthy's paper [mcca78] summarizes the initial development of LISP. As he notes, it began to rapidly diverge after a few years. Today, I think it is safe to say that there are two main dialects of LISP:

1. INTERLISP, which runs on DECSystem 10/20 machines, DEC machines, and Xerox 11xx Scientific Information Processors. A subset of INTERLISP, available from Uppsala University, Sweden, runs on IBM 30xx processors under VM/SP.
2. MACLISP, developed for the DECSystem 10/20 family by MIT's Artificial Intelligence Laboratory, which also runs on VAX-11/7xx machines. It, too, has spawned new machine architectures, principally Lisp Machine's LAMBDA machine and Symbolics' 36xx family of LISP Workstations. MACLISP has actually been superseded by CommonLisp, but the heritage remains the same.

In this section, I will briefly review and note some of the other LISP dialects and their major differences.

1.2.1 MACLISP/ZetaLisp

MACLISP is a dialect of LISP that was developed at MIT's Laboratory for Computer Science (formerly Project MAC). It runs on DECSystem-10/20s and VAX-11/7xx series processors. MACLISP is oriented to efficient numeric computation within a symbolic environment. MIT's Laboratory for Artificial Intelligence

pioneered the concept of personal computers running LISP through the development of the CONS (1976) and CADR (1978) Lisp machines. Subsequently, Symbolics [symb83] was formed to develop commercial Lisp machine products. Their first entry was the LM-2 which was a repackaged and enhanced CADR machine. In 1982, Symbolics announced the 3600, a fourth generation (they claim) Lisp machine.

Winston and Horn [wins81] introduce MACLISP in the first part of their book. The second part gives numerous examples of LISP applications drawn from the AI arena. Charniak, Riesbeck, and McDermott [char80] provide an extensive discussion of MACLISP features in their book along with numerous examples of AI programs.

There are major differences between MACLISP and INTERLISP. Winston and Horn note some of the most important differences are as follows:

MACLISP does not distinguish case information; every character is translated to upper case on input.

Several functions, such as SETQ, DIFFERENCE, and QUOTIENT, may take multiple arguments in MACLISP.

MACLISP has only three types of function evaluation as opposed to four in INTERLISP. These are related as follows:

MACLISP	INTERLISP
EXPR	EXPR
----	FEXPR
LEXPR	EXPR*
FEXPR	FEXPR*

MACLISP uses the function DEFUN instead of DEFINEQ. The syntax is somewhat different.

In MACLISP, MAPCAR's arguments are ⟨function⟩ and ⟨list⟩. MAPCAR may also take multiple lists as its arguments where the ⟨function⟩ is applied to each element of each list.

The last two arguments of PUTPROP are reversed in MACLISP. That is, the syntax of PUTPROP is

```
(PUTPROP ⟨atom⟩ ⟨value⟩ ⟨property⟩).
```

MACLISP comments start with a ; which may occur anywhere within a program without affecting the execution of the program.

Bitwise logical operations are performed by the function BOOLE in MACLISP with a numeric argument indicating which function is desired.

For their Lisp machines, Symbolics uses a variant of MACLISP known as ZetaLisp [wein81] which introduces the following new features [symb83]:

A full range of data types
Flexible function calling and multiple-value returns
Stream-oriented input and output
The Flavor System for object-oriented programming with message passing
Macros for extending ZetaLisp syntax
Predefined functions for sorting, hashing, linear equations, and matrix operations
Multiple name spaces (packages)

ZetaLisp will be compatible with CommonLisp, which has been developed by a consortium of academics and AI researchers along with considerable industry participation [stee84].

1.2.2 FranzLisp

FranzLisp was developed at the University of California, Berkeley by John Foderaro [fode81]. It runs under Berkeley Unix BSD 4.1 and Eunice on VAX-11/7xx series machines. It is being ported to a large number of microcomputers (primarily 16-bit machines) that also run BSD 4.1. FranzLisp is a subset of MACLISP. Lately, it has taken a development path that diverges from MACLISP through a series of enhancements designed and implemented by several universities notably the University of Maryland and Carnegie-Mellon University.

1.2.3 Portable Standard Lisp

Portable Standard LISP (PSL) is a version of LISP being developed by the Utah Symbolic Computation Group. It has been described in a series of technical reports by Griss [gris81, gris82a, gris82b] and Marti [mart79]. PSL is written in a language called SYSLISP whose definition is given in Benson and Griss [bens81] and Griss and Hearn [gris81]. PSL was developed to support the porting of the REDUCE-2 symbolic algebraic manipulation system to a number of different machines. It is rumored that PSL will be the Lisp of choice for the Cray machines.

1.2.4 CommonLisp

CommonLisp [stee84] is a collaborative effort of over 50 researchers and LISP programmers. Its authors claim that is a new dialect of LISP which is a successor to MACLISP, but which has been strongly influenced by Lisp Machine LISP, SCHEME, and less so by INTERLISP.

CommonLisp is intended to be a common LISP. That is, it should serve to unify the features of many of the successors to MACLISP that have been developed for a variety of machines. Several groups have indicated that they will ensure compatibility with CommonLisp once the standard reference manual is published.

CommonLisp is intended to ensure portability of programs among a large number of machines by eliminating those features that cannot be implemented easily on any one of the machines. Some details are left to the implementors and some features are made optional. The goal is to ensure that programs adhering to the CommonLisp standard will be transferrable among machines as long as they contain no machine-specific features.

CommonLisp is intended to define an internally consistent semantics for many standard LISP functions. Different implementations of LISP have treated variables in different ways which have resulted in different results when the program is moved from one machine to another (or even from one implementation to another, e.g., shallow versus deep bound INTERLISP systems). The reference manual specifies one and only one semantic interpretation for each function.

CommonLisp follows the MACLISP tradition in emphasizing the power of system-building tools. Its designers envision, much like MACLISP and INTERLISP, that many user-level packages will be built on top of CommonLisp, but these are not part of the core specification.

CommonLisp is designed to be efficient and stable. It is primarily compatible with MACLISP and its descendents/variants, but less so with INTERLISP.

CommonLisp will unify a community of LISP programmers who have become quite fragmented by pursuing variations of MACLISP. The approach is laudable in that LISP, at least the MACLISP side, is one of the last languages to develop a coherent standard. I encourage you, when writing programs in INTERLISP, to pay careful attention to the CommonLisp constructs in the event that you want to translate your programs to that dialect. Having translated several programs from esoteric implementations of FranzLisp, I can assure you that the reverse translation is quite painful and labor-intensive (particularly to debug).

1.3 LISP AS A PROGRAMMING ENVIRONMENT

LISP may be considered from two viewpoints: first, as a programming environment, and, second, as a conceptual environment. In this section, I will discuss LISP as a programming environment. In the next section, I will discuss it from a conceptual viewpoint.

Different application domains place different requirements on the language(s) which may be used to write programs for them. Put another way, no one language is suitable for all applications. LISP was developed to support artificial intelligence (AI) research and has become the primary tool for developing programs in that discipline.

This development has been neither circumstantial or serendipitous. LISP incorporates many ideas (indeed, it pioneered quite a few) that are now considered to be essential elements of the professional programming environment. The following paragraphs summarize the major requirements for an AI programming language and discuss how LISP satisfies them.

1.3.1 Multiple Data Types

AI systems are generally large, complex programs. Many types of information are used and need to be represented within the program. A variety of data types, with associated operators, make it easier to encode and manipulate information within a program. INTERLISP provides numerous data types (see Chapter 2). One of the major features of INTERLISP is that any data type may be passed as an argument to a function. In most cases, the function must decide what it has and how to process it. In addition, you may also define your own data types (see Chapter 27) and utilize system functions to operate upon them.

Lists, in one form or another, are the basic data structure used in AI programs. Facilities for the efficient manipulation of lists are very important. Obviously, INTERLISP satisfies this requirement as it is list-based. Chapters 3 and 6 discuss the basic list processing primitives.

As INTERLISP evolved, new data types were added to the language to increase its flexibility. Initially, INTERLISP supported only atoms, numbers (a special type of atom), and lists. Later, strings and arrays were added. Today, INTERLISP-D supports a number of basic data types such as bit maps and windows, to provide a powerful, window-oriented programming environment. This trend is significant because it makes INTERLISP useful as a general-purpose programming language.

Defining new data types, such as relational tables for a database application, is relatively simple. Providing the associated support mechanisms may or may not be so simple depending on the complexity of the data structure. However, the ability to develop a data type to meet your application needs without the artificial constraints that are often imposed by conventional programming languages should allow you greater flexibility in your programming.

1.3.2 Modular Programming

To make large systems understandable, you must be able to decompose them into smaller modules that are both readable and maintainable. Segregation of functionality makes a system easier to manage when updates or modifications are required. Traditional languages may support two or three methods of decomposition. For example, FORTRAN has both functions and subroutines. A simple model is necessary to minimize sideeffects while making changes an easy task. INTERLISP uses a functional programming model where the basic units are functions. A system is just a collection of functions that invoke one another.

One decomposition strategy is to gather common useful patterns of statements into a package. These packages are not just abbreviations to save typing (like macros). Rather, they are abstractions which encapsulate higher level concepts that have meaning independent of their implementation. Once a package is constructed and properly tested, a programmer may use it directly, without regard for its internal details, because it corresponds to a concept or primitive notion that he uses to solve the problem.

INTERLISP naturally supports the top-down programming methodology. You may define functions with stubs for functions to be called but defined later. The stubs are merely definitions of the functions which return NIL. When you are ready, you may use the editor to replace a stub by its full definition. The ability to incrementally build a system and test while you go has been called "prototyping" by certain programmers. You should note that it does not correspond to the classical top-down methodology espoused by most software engineers.

1.3.3 Deferred Binding

By binding, we mean the association of a value with a variable. For most AI programs, values are not known when the program is written, but become known as the program is executed. Conventional languages bind either at compile time or link time or, rarely, at execution time. INTERLISP binds at execution time. For example, the size of a list need not be known until some function is applied to it. The same is true for the number and types of properties associated with a particular atom. Moreover, the structure of a list need not be known until it is operated upon by other functions. Indeed, the list may carry a description of its structure with it, a feature that is difficult to replicate in conventional languages. The ability to break apart and put together lists with little or no regard for storage allocation is a feature that is rarely, if at all, found in more conventional languages.

1.3.4 Interactive Development

Most conventional languages are batch-oriented even though they may be accessed from a time-shared environment. That is, a complete program unit is submitted for compilation, then loaded, and finally executed -- where each operation is an independent activity. A few conventional languages such as BASIC and APL may be interactively executed.

INTERLISP allows you to begin execution of programs that are only partially written. At worst, attempting to execute a missing function generates an UNDEFINED FUNCTION error. More likely, it will cause a break (see Chapter 20) and allow you to attempt to correct the problem. The correction may be as simple as correcting a misspelling of the function name or defining the body of the function. Returning from a break usually allows you to resume execution with no detrimental consequences to your environment. While this is not an

ideal way to develop a program, it does provide a convenient mechanism for testing a set of functions after they are written without having to complete the whole program.

1.3.5 Flexible Control Structures

A control structure determines the sequence in which program statements and functions are executed. INTERLISP supports the standard control structures such as conditional execution, case selection, and iteration. It also provides mechanisms for recursion (as an inherent language feature), and coroutines for parallel processing. Beyond these **language-based** features, we may define more sophisticated control structures that facilitate data-driven or goal-directed behaviors. These control structures are usually embedded in expert systems. Three types of control structures that are often implemented in expert systems include [aiel83]

Goal-driven (backchaining)
Event-driven
Model-driven

Goal-driven strategies use a goal rule which invokes all rules whose conclusions are referenced by conditions in the goal rule. These rules invoke relevant rules in a chain until the rules to be executed reference only input data to the system.

Event-driven strategies use a set of inputs to determine the invocation of one or more rule-sets. Executing these rule-sets generates new events which invokes more rules and so on until some conclusion is reached (e.g., no more events may be generated using the given rule-sets).

Model-driven strategies match a current "state of the world" against problem models to generate expectations. Expectations are events that stipulate inputs to be looked for or requested from the user. As data are entered, the state of the world is updated until no more expectations may be generated (e.g., the model is complete within the given rule-sets).

While these strategies are presented within the framework of expert systems, I believe that they have general utility to other applications that might be written in LISP. Numerous books on AI, including Waterman and Hayes-Roth's book [wate78], discuss various approaches to these control structures.

1.3.6 Pattern Matching Facilities

Pattern matching is a content-based selection strategy where the alternative to be executed is not known until run time. INTERLISP has been utilized to build a number of different types of pattern matching programs. Kornfeld [korn79] discusses a class of high level languages known as pattern-directed invocation

languages. Waterman and Hayes-Roth [wate78] edited a book that describes recent (at that time) work in pattern-directed inference systems.

Patterns are merely templates for interpreting (or validating) otherwise arbitrary data structures. For example, the pattern (A ? ? D) matches the data structure (A B C D) but does not match the data structure (E ? ? D). Pattern matching allows us to retrieve a selected set of values of data structures (often called facts or assertions) from a larger collection by applying a pattern to them. Simple pattern matchers are easy to build and are often given as introductory programming projects in LISP courses. Kornfeld describes a simple pattern matcher and demonstrates its actions with a few examples.

In most databases, there are a number of facts that are not physically present in the database. More sophisticated pattern matchers have been developed that can deduce or infer data from existing data given a set of patterns to work with. These pattern matches form the basis for production rule systems, many of which are described in Waterman's book. Shapiro [shap79], Winston and Horn [wins81], and Charniak et al. [char80] all describe examples of simple deduction and inferencing systems based on pattern matching.

1.3.7 Language Extensibility

As I mentioned above, it is relatively easy to implement a very small kernel of LISP that allows you to "start" programmming fairly quickly. Rather quickly, you will find that you need additional functions or capabilities to implement problem solutions. Unlike most other languages, you will find that it is easy to extend the LISP language by merely introducing new functions. Many of the functions that you will encounter in this text have been written using basic LISP functions. That is, they represent extensions to the LISP language. The flexibility and ease with which you may extend the language is perhaps its most powerful feature. This means that you may tailor the form and features of the language to any class of users or any type of application. Once you begin programming in LISP with any regularity, you will find that you think in terms of language extensions rather than just implementing programs.

1.4 LISP AS A CONCEPTUAL ENVIRONMENT

Program design is a creative (some say artistic) process. Humans solve the problems while computers implement the solutions. But the transformation from concepts to executable code is often a difficult and laborious process. The variety of problems to be solved means that there can be no standard recipes or mechanized processes to effect program design. For many problems, we do not know the nature of the solution although we may have a good idea of what the answer should be. The programming process then becomes one of exploration and experimentation with solution techniques to elicit the essence of the problem. Of-

ten, the solution emerges along with the understanding of the problem. This method of programming is sometimes known as **prototyping**.

LISP provides an environment that makes prototyping an effective means of exploring the solutions to problems. What we look for is the conceptual architecture of the problem, the steps to be taken to "solve" the problem, e.g., to produce the right answer. Using conventional languages, we almost always must know the entire solution before we begin to write the program. With LISP, we can incrementally build and refine the solution as we explore different alternatives in the problem representation.

Stefik et al. [stef82] note that design is the making of specifications to create objects that satisfy particular requirements. The problem that faces the programmer is to develop the specifications for the program that produces a specific result. However, when the algorithm(s) and methods for solving the problem are not explicitly known, conventional languages offer no help in designing the program because they require us to have a complete conceptual architecture before coding.

Stefik et al. note that there are five key problems associated with design:

1. In large problems, a designer cannot immediately assess the consequences of design decisions. He must be able to explore design alternatives tentatively.
2. A design will be constrained by many sources. Oftentimes, the constraints are imposed by the features and capabilities of the language in which the solution will be implemented.
3. To solve large problems, the designer must cope with system complexity by factoring the design into subproblems. He must also cope with interactions between the subproblems, since they are seldom independent. Consider the simplicity (after studying the LISP language) with which you may use top-down programming to decompose a complex problem into simpler, more manageable pieces.
4. When a program becomes very large, it is easy to forget the reasons for design decisions. It is also hard to assess the impact of a change to a part of the program. While LISP does not provide any mechanism for capturing design decisions (other than comments), the ability to immediately test modifications to program segments and repeal (undo) those which do not lead to progress in the problem solution is a feature that is not often found in other languages (or is very difficult to perform).
5. When programs are modified to reflect new design decisions, it is important to be able to reconsider different possibilities. Few languages have the inherent capability to maintain multiple versions of functions or to manage the structure of programs spread across one or more files. Through the File Package, INTERLISP provides this feature as a standard capability.

1.5 STRUCTURE OF THE TEXT

This text is divided into 31 chapters, a reference list, and an index. It loosely follows the organization of the INTERLISP Reference Manual [irm78, irm83]. However, I have taken the liberty of organizing the contents of individual chapters according to my own logical view even though this contravenes the IRM.

Chapter 1 is an introductory chapter. It contains a short history of LISP and a discussion of different LISP dialects. LISP is examined in both the programming environment, as a language for implementing tools and systems, and in the conceptual environment, as a way of thinking about the functional architecture of systems. The remainder of the chapter contains this outline and notes concerning the presentation methods within the text.

Chapter 2 discusses the basic data types provided by INTERLISP. These include atoms, lists, numbers, strings, and arrays. Operations on these data types are described in succeeding chapters.

Chapter 3 discusses the primitive functions of INTERLISP. Most other functions can be built using these primitives. These functions are usually **hardwired** in machine language or microcode for purposes of efficiency. In general, most of the functions defined in this chapter and Chapters 4-13 are common to all dialects of LISP although there may be a few discrepancies. You should note that INTERLISP is generally much richer in its primitive functions than most other dialects of LISP.

Chapter 4 discusses the fundamental predicates. Predicates are a means of testing the condition of an S-expression or atom (or other datatype). A predicate returns either NIL (meaning false), T, or some non-NIL value (both of which mean true). Most of the fundamental predicates test whether their argument is a given datatype, whether the argument has a certain characteristic (e.g., evenness or oddness) or whether two things are equal.

Chapter 5 discusses logical connectives and predicates. A logical connective is either AND or OR. Logical predicates are EVERY, SOME, and their variants. Note that it is possible to implement sophisticated control structures using the AND and OR functions.

Chapter 6 discusses the list manipulation functions. These functions allow you to create and destroy lists, concatenate two or more lists, and copy lists. Together with the functions presented in Chapters 3 and 4, these functions usually constitute the basic functions of the Lisp language.

Chapter 7 discusses property lists and functions for manipulating them. Property lists are attached to atoms. A property list is a list of ⟨name⟩ ⟨value⟩ pairs which may be thought of as descriptors or characteristics of the atom.

Chapter 8 discusses function definition and evaluation. A function is a specification for operating upon an atom, list, or other datatype or data structure. INTERLISP provides several methods for defining and using functions.

Chapter 9 discusses atom manipulation and the rules for creating atoms. Atoms are the elementary information carriers in INTERLISP.

Chapter 10 discusses string manipulation functions. Strings represent an unusual datatype in that they are neither atoms nor lists.

Chapter 11 discusses arrays, including hash arrays, and their associated functions. It describes how to create 2-D arrays, known as matrices.

Chapter 12 discusses the mapping functions. Mapping is a technique for iterating (applying) a function over multiple instances of a datatype.

Chapter 13 discusses the basic arithmetic functions, both integer and floating point. Special functions, including some statistical functions, are also described.

Chapters 14 and 15 describe the functions associated with input and output. Rather more information is provided in these chapters than the novice programmer needs to know. Judicious reading will help you decide what is immediately necessary. Mastery of the input/output functions is essential to becoming a competent INTERLISP programmer.

Chapter 16 discusses file management functions. Unlike conventional languages, which depend on the operating system or system utilities for file management [kais82], INTERLISP incorporates a set of file management functions into its environment. Depending on the environment, these functions may or may not interface with external operating system functions.

Chapter 17 describes the features and capabilities of the File Package. At one level, the File Package may be viewed as a sophisticated source code control system similar to that provided by the Programmer's Workbench in UNIX [dolo78].

Chapters 18 and 20 discuss error handling and debugging procedures. Whenever an error occurs, INTERLISP allows you to inspect the state of your environment, modify it, and (hopefully) resume your computation.

Chapter 19 discusses the LISP editor. The editor understands the structure of S-expressions and function definitions, so it is known as a **structure editor**. It operates upon S-expressions in a manner that is (usually) guaranteed to preserve the balance of parentheses. Novices may begin with a few simple commands. Experienced users have a wealth of powerful, flexible commands at their beck and call.

Chapter 21 discusses the process of advising a function. Advising allows you to modify the interface between a function and its environment, whether or not the function is called from another function. You may "intercept" the function before or after it is executed.

Chapter 22 discusses the DWIM (Do What I Mean) facility. DWIM provides an error correction facility for simple errors associated with spelling, and a facility for handling simple structural errors. DWIM receives control before the error mechanisms described in Chapters 18 and 20.

Chapter 23 discusses Conversational LISP (CLISP). CLISP allows you to use English-like phrases in place of some of the normal INTERLISP constructs. CLISP makes LISP programs appear more like a conventional language and hence more readable.

Chapter 24 discusses a number of LISP users packages. These are packages or subsystems contributed by various INTERLISP users that extend the capabilities of the system in various ways. As INTERLISP-D continues to spread throughout the AI and programming communities, more packages are being developed and submitted to Xerox all the time. Xerox has incorporated some of the most useful and practical packages directly into the INTERLISP-D system.

Chapter 25 discusses the Programmer's Assistant. The Programmer's Assistant performs simple error correction, including spelling correction, on the more obvious errors committed by a user when typing in a function or S-expression to the system.

Chapter 26 discusses Masterscope, a utility for managing aspects of the code development process. Masterscope allows you to keep track of the modules or functions that have been developed, to examine their interactions, and to maintain (to a limited extent) subsets of the functions comprising your program.

Chapter 27 discusses the Record Package. The Record Package effectively implements a data-access programming technique that relieves you of the burden of worrying about storage management. You determine the data structures and how they are to be used. INTERLISP translates these declarations into calls to primitive functions when it executes your programs.

Chapter 28 discusses the History Package. Together with the Programmer's Assistant, the History Package provides a flexible environment for developing source code. The History Package effectively maintains an audit trail of your interactions with INTERLISP at the top level. You may retry, correct, or undo erroneous expressions, and you may create new commands within the package.

Chapter 29 is a catchall chapter for functions that don't seem to have a logical fit elsewhere. This is really not a fair interpretation of the utility of these functions, but they are too few to warrant a chapter of their own. Included are the clock functions, greetings, storage management, and a function for regaining space when your virtual memory becomes full.

Chapter 30 discusses the structure of the underlying INTERLISP environment. INTERLISP dynamically binds values to variables using a stack mechanism. The stack is accessible to you through stack manipulation functions or through the Break Package.

Chapter 31 discusses some aspects of the process of compiling. Once you have debugged your programs to the point where you think they can run without major errors, it is often advisable to compile the code in order to gain increased efficiency and performance.

1.6 COMMENTS ON THE TEXT

The structure of this text reflects some of my personal biases about how to present material to a technical audience. It is important that you, the reader, be aware of these biases as you read (or peruse) this and succeeding volumes.

One major bias is that readers understand and (hopefully) learn more about the subject by seeing many concrete examples of the things that are being talked

about. You will find this text replete with examples for the many functions and features that are discussed. In addition, I show how to develop new functions for capabilities that are not currently provided by the baseline version of INTERLISP (although this is changing as a result of Xerox's continued development and enhancement of the language and programming environment).

Another bias is that people learn more about how things work by seeing how functions and programs are put together. Thus, in many cases, I have tried to show the basic code that is used to implement a feature. The code is often a skeletal representation of the functions necessary to implement the full capability. Some functions are merely suggested while others are developed in full.

Where functions take multiple arguments, I have used a tabular format to show the structure of the function and describe its arguments. I feel this makes it easier to refer to the meaning and/or usage of arguments that may be given to a function.

I have tried to follow several conventions in this text to make it easy to read. These conventions are listed below:

When I reference an INTERLISP function, whether in text or in an example, it will always be capitalized.

Arguments to INTERLISP functions, whose skeletons are given in the text, will always be entered in lower case, except when upper case is specified explicitly in the INTERLISP Reference Manual [irm78,irm83].

When I refer to arguments of INTERLISP functions in text, I will capitalize the names of the arguments so that their usage is clear.

When I define a function, I will capitalize the primitive functions used, but generally show the function definition in lower case. I believe this highlights the distinction between the basic functions and those that may be developed "on top of" INTERLISP. I also believe that it improves readability of the text. You should note, however, that INTERLISP always operates in upper case.

I have consistently tried to use the symbol ← to indicate the prompt throughout the text. This symbol is normally used only on INTERLISP-D. I have represented it as two characters since most terminals (and typesetters) do not have the left arrow as a standard symbol. Please be aware that left arrow has several other meanings as well in INTERLISP.

2

LISP Data Structures

After you have defined a problem, one of the first tasks that you face in capturing the problem solution in executable code is how to represent the data required to solve the problem in a form manipulable by the computer. In doing this, you are constrained by the data types and structures provided by the programming language. In most conventional programming languages, these are fixed by your choice of language. In INTERLISP, a set of basic datatypes is provided with the language, but you are also provided with the ability to extend these through various mechanisms inherent in the language.

INTERLISP provides a wide variety of datatypes. The primitive datatypes are lists and atoms. Other datatypes have been provided to increase the efficiency of INTERLISP and remedy the problems of representation of certain types of data structures. This chapter will describe the basic datatypes used by most INTERLISP programmers. Extensions to these datatypes will be discussed in chapters that describe the functions for manipulating the datatypes.

2.1 LITERAL ATOMS

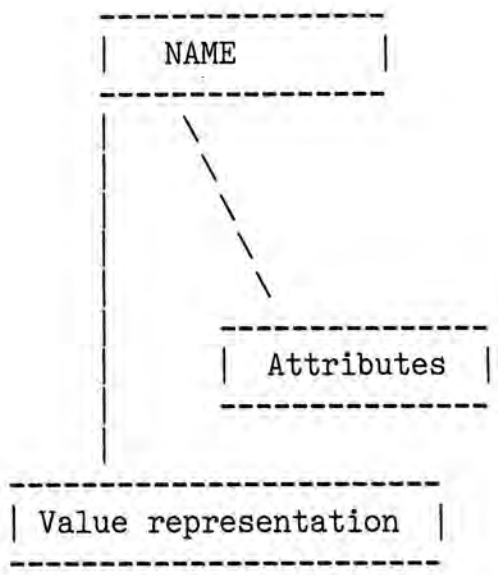
A **literal atom** is the most basic datatype of INTERLISP. A literal atom is something that stands for itself. Thus, every literal atom created in the INTERLISP system must be unique. If two literal atoms have the same name (called the **printname** or **pname**), then they are the same identical atom. This means that these atoms will always have the same address in memory.

Atoms play the part of variables in INTERLISP. Different programming languages treat the notion of variables in different ways depending on how the variable is stored in memory and how values are bound to it. Variables allow you to write programs in which the actual values of data items are not known until the program is executed. However, by implication, the values associated with variables in a programming language are always finite because of the limitations of the computer.

The "attributes" of a variable describe how the variable will be treated in a program. Among the attributes that may be associated with a variable are

The permitted range of values
The scope of its name
How and when it is created
The type of the variable

A simple model of a variable might appear as:



Unlike conventional programming languages, INTERLISP variables do not have an explicit type associated with them. Rather, the variable takes on the type of its current value whether it is a array pointer, integer, string, or another atom.

The "value representation" describes the form that the value of the variable takes in memory. Often, the value representation is a function of the underlying hardware. However, list representation, even though an elementary datatype, is a data structure imposed on top of (or extending) the essential machine hardware.

INTERLISP extends this simple model with additional information. Each literal atom consists of the following components

a pname (print name)
a value
a property list (see Chapter 7)
a function definition cell (see Chapter 8)

2.1.1 Pnames

A **pname** is the name of the atom as it appears when printed by one of the INTERLISP printing functions. Pnames are not directly accessible by the user. The representation of a pname in memory is dependent on the particular implementation. The maximum length of a pname is a 127 characters for INTERLISP-10 and 255 characters for INTERLISP-D.

Examples of atoms include the following

```
BEN FRANKLIN        COUNTRY
1.2567              751
```

2.1.2 Value Cells

A **value cell** is a storage area assigned to hold the value of an atom. A value cell always contains the top-level value of an atom. When an atom is initially created, the value cell contains the atom NOBIND which indicates that no explicit value has been bound to the atom. Atoms may also have values bound in stack frames as the result of PROG expressions and function calls.

A value cell is created for an atom when

The atom is referenced in a SETQ function

The atom is used as the argument to a function

The atom is used as a PROG variable (see Section 3.7)

Values are really pointers to other INTERLISP objects rather than the actual values themselves.

Binding is discussed in Chapter 30.

2.1.3 Property Lists

Each atom may have a value. The value may be a number, another atom, a list, or an address of some other structure, such as an array. But atoms really represent (in most cases) complex objects; that is, they stand for something. These object usually have a multitude of characteristics. Most conventional languages require you to define and manipulate several complex data structures in order to fully define an object. LISP, however, allows you to attach a property list to an atom. A **property list** is a list of descriptors and their associated values which serves to further define or describe the object represented by the atom.

2.1.4 Function Definition Cells

Each atom has an accompanying function definition cell, e.g., a cell whose contents are a list that defines a function of the same name. Thus, in INTERLISP, atoms may have both values and actions associated with them. INTERLISP dis-

tinguishes between these two usages by the appearance of the atom name in a list.

2.1.5 Creating Atoms

There are several ways to create atoms in INTERLISP:

by assigning a value to a literal atom's pname
by packing a sequence of characters to form an atom pname
by making an atom
by asking the system to generate a temporary symbol

Setting the Value of an Atom

An atom may be created by assigning a value to a literal atom. This is the most commonly used method of creating atoms in INTERLISP. As an example, consider the assignments

```
(setq boy 'john)
(set 'boy 'john)
```

Both of these statements have the effect of assigning the value "john" to the atom whose name is "boy." The difference results from the operation of the function SETQ (see Section 3.8). If this is the first time that "boy" has ever occurred in an INTERLISP program, when the function (either one) is executed, the atom "boy" will be created and assigned the value "john." By created, we mean that INTERLISP allocates memory for it, defines the underlying data structure, and places JOHN in the value cell of BOY.

Creating an Atom via PACK

An atom may be created by packing a list of atoms to form the PNAME of a new atom. Typically, the list consists of atoms each of which is a single character.

Consider the list (b a l t i m o r e). When this list is given to PACK (see Section 9.3.1) as an argument, it concatenates the individual characters into a PNAME that represents a single atom. In this case, the single atom is BALTIMORE. Of course, the list does not have to be composed of single characters as in the following case:

```
(PACK '(tom-jones))
```

which yields the atom TOM-JONES.

Making an Atom

An atom may be created by the function MKATOM which operates on strings (see Section 9.2.2). For example, the function call

```
(MKATOM "john-jones")
```

would create the atom JOHN-JONES.

Generating a Temporary Atom

Atoms may be created by INTERLISP at your request. INTERLISP provides the function GENSYM to create atoms of the form xnnnn where x is a character and the n's are digits. Typically, atoms generated in this way are used for temporary storage within an INTERLISP program.

To create an atom using GENSYM, you may either specify the character or NIL as in the following examples:

```
(GENSYM 'B)     which produces an atom such as B0102
(GENSYM)        which produces an atom such as A0001
```

A counter is maintained internal to INTERLISP. The value of this counter is used to determine the digits which are appended to the character passed to GENSYM. If no character is specified by the user, INTERLISP automatically assumes the character A.

2.1.6 Binding Variable Values

Binding is the act of associating a value representation with a variable. There are two notions of binding that you must understand in INTERLISP:

1. Bound versus unbound atoms

A "bound" variable has a legal value assigned to it. An unbound variable has no value assigned to it (not even NIL). If you use a variable in an expression, INTERLISP attempts to evaluate it, finds that it does not have a value, and issues the error message "U.B.A." for "UnBound Atom". Atoms created without values (such as in PROG declarations) will have the atom NOBIND placed in their value cell. NOBIND is a special atom within the INTERLISP system which indicates "no value." A predicate, BOUNDP, allows you to determine whether or not an atom has a value bound to it.

2. Global (free) versus local atoms

INTERLISP programs (systems) consist of a set of functions that are linked together by calls from one another. Variables may be global or local within a set of functions. A local variable is one which is defined within a function and is used only within that function.

```
Typically, local variables are declared by the LOCALVARS
declaration or by appearing in a PROG header. Global
variables are variables defined in one function and used
(freely) in one or more other functions. The INTERLISP
name space is a one-level name space. Thus, any atom
defined by a SETQ, GENSYM, etc. is known to all
functions.
```

Another type of binding is the time when storage is allocated to the variable. Most conventional languages and systems will bind variables to memory locations at one of three times:

Compile time
Link edit time
Execution time

INTERLISP dynamically allocates storage to all atoms at execution time. This allows memory to be used efficiently in the computation process at some expense to performance in managing the memory. Currently, one of the critical problems with INTERLISP (and other LISP dialects) is that once atoms are created they cannot be destroyed in the same session. Variables created in a PROG header are temporary place holders on the stack. All other atoms are allocated specific storage locations and become a "permanent" part of your INTERLISP environment. Methods for managing your environment to save only the relevant information are discussed in Chapter 17.

2.1.7 Variable Typing and Declaration

Most conventional languages require you to declare your variables before you use them. That is, you must "announce" the name and type of a variable to the compiler in a declaration statement before the variable name is used in an executable statement elsewhere in the program. The "type" of a variable determines

The set of values it may assume
The set of operations to which it may be subjected

For example, declaring a variable to be a STRING (or CHAR) specifies its storage representation and the permissible operations upon it. Thus, I can concatenate two strings, but I cannot add them together. Each TYPE in a conventional language has a set of operations that are checked by the compiler at compilation. The compiler generates error messages if you attempt to violate the type of a variable. Note that the type of the variable is (usually) permanent throughout the program.

INTERLISP (and other LISP dialects) do not require you to declare or type a variable. A variable is declared when it is referenced for the first time. Its type is determined when you make an assignment of a value to the variable. Thus, the "type" of a variable may change during the execution of a program as different values are assigned to it. Type checking is performed when the value of a variable is utilized since a function cannot know the type of the variable's value until it has actually referenced it.

The notion of typing leads to the concept of "constants," e.g., data items whose value cannot be changed. INTERLISP (and other LISP dialects) do not utilize the notion of constant except in the following cases

The atom T has a fixed value that may not be changed.

The atom NIL has a fixed value that may not be changed.

Numbers (both FIXP and FLOATP) have values that may not be changed.

In some systems, NOBIND is represented as an atom whose value may not be changed.

Thus, aside from these few cases, any atom may be assigned any value at any time during execution.

2.2 NUMBERS

All programming languages include the capability to manipulate numeric data. Even in nonumeric data processing, there is a need to use numbers as counters, system parameters (such as line length), and control variables in a computation. Unlike mathematics, however, the classes of numbers that we use in programming languages are finite. That is, the range of values of a class of numbers is constrained by the physical attributes of the machine (such as register size).

INTERLISP supports two types of numbers: integers and floating point numbers. Every number is an atom. They differ from literal atoms in that they do not have property lists, value cells, function definitions, or explicit pnames.

2.2.1 Integers

An integer is represented as a string of digits preceded by an optional sign (either "−" or "+") and followed by an optional "Q". If the Q is present, the number will be interpreted as an octal (base 8) number. INTERLISP/370 allows you to enter hexadecimal numbers by entering an optional "−" or "+" followed by an "at sign" (@) followed by a string of hexadecimal digits (0 ... A ... F).

Integers may be either "small" or "large." Small integers fall into the range [−65536,65536] while large integers fall in the range [−2**32,2**32]. For INTERLISP/370, the range of small integers is [−2**23,2**23] which fits into the 24-bit address field used by IBM Series 370 machines.

Integers may be created by PACK and MKATOM since they are atoms, as in the following example:

```
(PACK
     (LIST 3 4 5))
```

will create the integer 345. In a similar fashion, we could also specify the following code:

```
(PACK
     (LIST 7 7 'Q))
```

which would yield the integer 63. In this case, the 'Q means that the integer is 77Q (i.e., 77 base 8 or 63 base 10).

Small integers are given special storage in INTERLISP in order to increase the efficiency of the system. Large integers are not. Thus, the user should be careful when testing the values of integers. Two small integers are always guaranteed to have the same representation. Two large integers may have the same value but not the same address in memory. Thus, if you use the EQ function on the two integers, they may not be equal. Instead, use the function IEQP or EQUAL to test the equality of two integers (see Section 13.2).

2.2.2 Floating Point Numbers

A floating point number corresponds to a real number in traditional languages such as FORTRAN. A floating point number is read or written as a (possibly) signed integer, followed by a decimal point followed by another sequence of digits. The digits following the decimal point are known as the **fraction** or **mantissa**. A floating point number may be optionally followed by an **exponent** designated by E followed by a (possibly) signed integer in the range -39 through $+38$.

Whether the fraction or the exponent is used is at your discretion, but one must be present to distinguish a floating point number from an integer.

Floating point numbers may be created during input (via READ), by applying FLOAT to an integer, and by PACK or MKATOM.

Floating point numbers are printed by the system using format controls specified by the function FLTFMT.

INTERLISP-D Convention

Floating point numbers are currently stored in single precision using the IEEE "single" mode of 32 bits.

INTERLISP/370 Convention

If the floating point number contains six or fewer digits, it is stored as a single precision number. If it contains more than six digits, it will be stored as a double

precision number. Floating point numbers in the range 1E−3 to 1E6 will be printed without an exponent; otherwise they will be printed with an exponent.

2.2.3 Complex Numbers

A **complex number** is a generalization of the real number that is introduced in mathematics so that all polynomial equations with real coefficients may have solutions. A complex number is composed of two parts: a **real** part and an **imaginary** part. Most conventional languages, developed for numeric data processing, support the notion of complex numbers. For example, FORTRAN has a type declaration COMPLEX which results in a specific storage representation and set of operations upon complex numbers. INTERLISP (and other LISP dialects) do not support complex numbers. However, it is not difficult to create a data structure to provide this capability. Chapter 13 will dicuss some ideas concerning the implementation of complex numbers.

2.2.4 Conversion between Numeric Classes

In mathematics, integers are a "subclass" of the real numbers. In computer systems, because integers and floating point numbers must be represented differently in storage, integers cannot be considered a subclass of floating point numbers. Thus, we need a set of conversion routines that allow us to convert data from one representation to another. However, conversion introduces inaccuracy in our numeric representations due to precision of representation and "roundoff" errors in arithmetic computations carried out by the machine.

2.3 LISTS

The second basic datatype within INTERLISP is the **list**. A list is a collection of zero or more components which may be atoms or other lists. A list with no components is often called the *empty* or *null* list. Lists may have atoms as their components, other lists as their components, or a mixture of both. A list is usually viewed as an "ordered" collection of components. The components may be integers, floating point numbers, strings, array pointers, or other data objects.

The format of a list begins with a left parenthesis (. It is followed by any number of atoms or other lists and is terminated by a right parenthesis). Thus, a very simple list would appear as (MARYLAND) where the atom's name is MARYLAND. Of course, the simplest list that we may write is ( ) which is a representation for the null list.

Examples of more complex lists include

```
(MARYLAND VIRGINIA WEST-VIRGINIA)
```

where individual atoms are separated by blanks and

```
(MARYLAND (CAPITOL ANNAPOLIS) (GOVERNOR HARRY-HUGHES))
```

which consists of an atom and two sublists represented, respectively, by

```
(CAPITOL ANNAPOLIS)
(GOVERNOR HARRY-HUGHES)
```

In general, lists are known as *S-expressions* where the S stands for "symbolic." Thus, the generic form of a list is given by

```
(S-expression[1] S-expression[2] ... S-expression[n])
```

The simplest form of an S-expression is the *dotted pair* which consists of two atoms, respectively the CAR and CDR. A dotted pair usually appears as

```
(apple . orange)
((the rain in spain) . (stays mainly in the plain))
```

The trend, today, is to discourage the use of the "dot notation" because it does not correspond to the recent implementations of LISP. I mention it here only so that you know of its existence.

The key to INTERLISP is parentheses. Lists are well-formed when pairs of left, (, and right,), parentheses balance in an expression. Many problems arise in writing INTERLISP expressions and functions because parentheses do not balance. Thus, a cardinal rule is to always check your parentheses (or use software that does so for you—see the INTERLISP Users Packages).

Lists may be created, destroyed, and manipulated in many interesting ways. We can create lists using CONS, LIST, or APPEND. We can access elements of lists using CAR, CDR, and their variants. We can manipulate them using RPLACA, RPLACD, ASSOC, MEMBER, and many other functions. Chapters 3 and 6 describe many of the primitive functions that operate upon lists.

We may constrain the full generality of these list structures in order to create other types of data structures. Some of these data structures include

Stack	A list in which all insertions, deletions, and accesses are made at one end of the list, known as the "top" of the stack.
Queue	A list in which additions can be made only at one end and deletions only at the other end.
Deque	A "double-ended" queue in which insertions, accesses, or deletions may be made at either end of the list.

Knuth [knut68], Horowitz and Sahni [horo82], and Aho, Hopcroft, and Ullman [aho83] all discuss the theory, representation, and manipulation of these data structures.

2.4 ARRAYS

Arrays are one of the most familiar data structures in programming, perhaps because we use tables (arrays) to store information in our everyday activities. An **array** is an ordered set of elements of the same type. Each array has *dimensionality*, the number of subscripts that must be used to reference an individual item within it. Vectors are one-dimensional arrays while matrices are two-dimensional arrays.

Vectors are often represented in the following form:

```
V = {v1, v2, v3, v4, ..., vN}
```

Matrices are often represented in the following form:

```
--                                  --
|    a11  a12  ...  a1N              |
|    a21  a22  ...  a2N              |
|    .                               |
|    .                               |      =    A
|    .                               |
|    .                               |
|    .                               |
|    .                               |
|    aM1  aM2  ...  aMN              |
--                                  --
```

An array is represented as a block of contiguous storage of arbitrary length. Array storage (in INTERLISP-10) is divided into three sections

A header

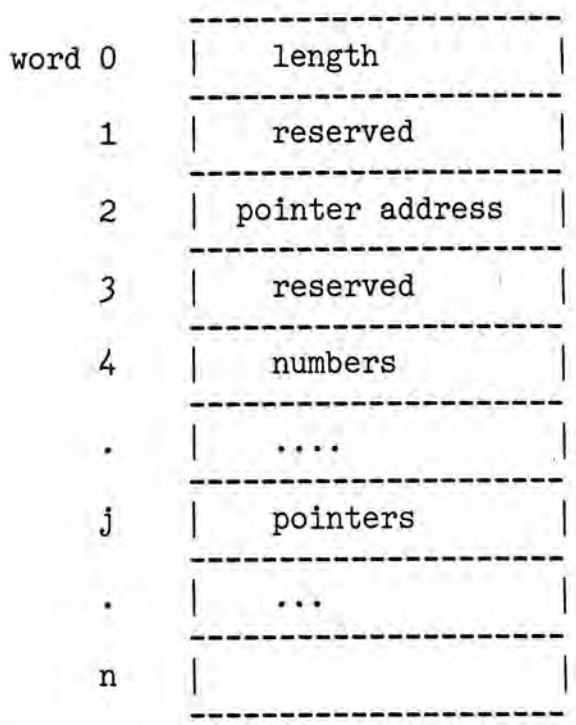
A section containing unboxed numbers

A section containing pointers

The header of an array contains descriptive information. Its format depends on the implementation.

The unboxed number section stores numbers, either real or floating point. The pointer section contains INTERLISP-10 pointers that allow an array to represent a collection of objects other than numbers. The size of the unboxed and pointer regions may range from 0 to the size of memory. Arrays are explained in greater detail in Chapter 11.

INTERLISP/370 Convention
INTERLISP/370 represents the internal format of an array as follows:



where the length of the block is given by

```
16 + 4 * ARRAYSIZE
```

The value 16 is derived from four bytes times four words of header information. Each number and/or each pointer also consumes four bytes.

INTERLISP-D Convention
INTERLISP-D has extended and refined the concept of arrays to fit its new implementation. In this implementation arrays contain only one "type" of data so there is no division of an array into partitions. Data types that may be stored in arrays include BIT, BYTE, WORD, FIXP (integers), FLOATP (floating point numbers), and POINTER. Arrays may have either a 0-origin or a 1-origin; the default is 1 as required by the TENEX implementation.

2.4.1 Dimensionality

The dimensionality of an array should be chosen to meet the needs of the problem to be solved. Most languages allow you to specify up to three dimensions (each of different size) for an array. Some languages allow an unlimited number of dimensions. Most applications usually require no more than three dimensions. Unfortunately, INTERLISP supports only one-dimensional arrays at the

current time. In Chapter 11, we describe a method for implementing two-dimensional arrays using the one-dimensional structures provided by INTERLISP.

The *bounds* of an array's dimensions correspond to the lower and upper limits on the values of its subscripts. Many languages allow these limits to be any pair of integers such that the lower limit is less than the upper limit. INTERLISP requires that all subscripts have a lower limit (the *origin*) at either 0 or 1. The lower limit must be the same for all dimensions of the array. This allows both conventional mathematical numbering as well as the practice that is more common in most conventional programming languages.

2.4.2 Specification and Creation of Arrays

Several pieces of data must be known to create an array (following the INTERLISP-D convention)

The type of the elements
The size or bound of the array
The origin of the array
The initial value of each element

Arrays may only be created via the ARRAY function (see Section 11.1). Elements may only be accessed via ELT and set via SETA. Unlike most conventional languages, INTERLISP allows you to specify an initial value that is assigned to every element of the array when it is created. Arrays are referenced in INTERLISP by passing the address of the array to various functions that can manipulate them.

2.4.3 Hash Arrays

A second type of array is the **hash array**. It is an array that provides a linkage between one INTERLISP data type, the *hash item*, and another data type, the *hash value*. A hash array consists of a number of cells defined when you create the hash array. To enter an item into the hash array, you provide both a key and its associated value. A hashing function (internally defined) is applied to the key to generate a cell index. A *hash link*, consisting of a pointer to the hash item and a pointer to the hash value, is placed in the cell. You obtain an item from a hash array by specifying its key.

When a hash array is seven-eighths full, INTERLISP assumes it to be completely full. Attempts to add new items to the hash array will result in an error. However, INTERLISP does provide a mechanism for handling these *overflow* conditions which is explained in Section 11.3.5.

Hash arrays are used by several of the INTERLISP packages. A notable instance is CLISPARRAY which is used by the CLISP package (see Chapter 23).

2.5 STRINGS

A string is a collection of alphanumeric characters that has a literal value but does not represent a data item. The number of elements in a string is called the *length* of the string. Strings are demarcated by pairs of " (*double-quotes*). Any characters except " and % may appear within a string. Clearly, " may not be included because it is the marker for a string. % has the special meaning of **escape** which is discussed in more detail in Chapter 14.

A string is not a fundamental datatype since it cannot be read directly by READ. Strings are created by the function MKSTRING, and manipulated by the functions SUBSTRING and CONCAT. Strings may be read in from external files via the function RSTRING.

A number of different operations may be defined for strings. The two most important operations are *concatenation*, where we join together two strings to make a new one, and *substring*, where we extract a segment of a string to make a new one. Most other string operations can be defined as sequences of these two operations.

It is also frequently necessary to scan strings for a specific character or sequence of characters. Scanning is utilized in many important applications (text processing, message decoding, etc.). It includes

Searching for delimiters

Searching for words in order to construct indices and concordances

Searching for character patterns in messages in order to recognize sections of the message

Scanning may be programmed with the substring access function, but most implementations of INTERLISP provide a search function to improve the efficiency of the scanning process.

Internally, a string is stored in two parts: a string pointer and the sequence of characters that compose the string. A string pointer consists of the length of the string and the address at which the string begins. INTERLISP-10 and INTERLISP-D both support 32,767 characters as the maximum length of strings.

INTERLISP/370 Convention
INTERLISP/370 supports only 256 characters as the maximum length of a string owing to machine instruction characteristics.

2.6 RECORDS AND USER DEFINED DATATYPES

The datatypes described in the previous sections are inherent features of a basic INTERLISP system. Users, however, may extend the classes of datatypes by adding new list structures. INTERLISP formally captures this notion in the Record Package which is described in Chapter 27.

A record is an object that has a formal description. Each record has a fixed number of fields. The record may be thought of as an object template which is used to create instances. A user may access and replace the contents of individual fields within an instance of the record. The individual elements of a record need not all be the same datatype. This notion is captured in many conventional programming languages in various forms.

Elements of a record are accessed by their name. You may reference the whole record or any part of it. Individual fields of the record may themselves be records (subrecords). Since a record description is a template for a data structure (e.g., an S-expression), we may lay the template over any S-expression.

In addition, you may define new datatypes with complex structures via the function DECLAREDATATYPE (see Section 27.7). The primary difference between records, which also use the datatype functions, and your own datatypes is that the File Package and other subsystems "know" about records and how to formally treat them. In most cases, you will have to provide your own functions for manipulating, saving, and restoring your datatypes.

2.7 FILES

Files are not strictly an INTERLISP datatype. However, most programs require a mechanism for specifying long term storage of data in an organized manner. The File Package supported by INTERLISP, which interfaces with the host operating system, provides a method for the user to retrieve, store, and organize external data.

INTERLISP treats files as a byte stream much like Unix. You may open a file and read from or write to it using various input and output functions. A hierarchy of capabilities is provided by the File Package which allows you to treat files in various ways. Since INTERLISP functions are just S-expressions, function definitions, data, and file descriptors (e.g., commands for creating the file) may be intermixed in a single file. This feature is unlike conventional programming languages which require the separation of the data and program code.

When a **symbolic file** is written by the File Package, a *file map* is placed at the end of the file. A file map contains the names and addresses (as byte offsets from the beginning) of objects within the file. Many of the File Package functions use the file map to extract objects and their values from a symbolic file.

One special type of file is written by INTERLISP—a file containing compiled code. Compiled code is a combination of low-level function calls and machine language instructions. Its format and contents depend on the implementation. Aside from a few examples, we will not discuss this type of file in this text because it is implementation dependent.

3

Primitive Functions

INTERLISP, as a symbolic processing language, encourages the use of "functional programming" concepts. Functional programming emphasizes the use of well-defined functions that "operate" on data structures or objects—either system defined or user defined. The previous chapter discussed some of the basic data structures provided by INTERLISP. This chapter explores the primitive functions that you may use to begin constructing more complex functions. Most primitive functions are "hardwired" in assembly language or microcode for reasons of efficiency.

3.1 TAKING LISTS APART: CAR AND CDR

INTERLISP provides two functions for taking lists apart; that is, decomposing a list into its constituent elements. These are **CAR** and **CDR**. The names are mnemonic and are rooted in the historical implementations of INTERLISP's predecessors. The earliest version of LISP was implemented on an IBM 704 computer in assembly language. The 704 word had two key fields: the address field and the decrement field. The CAR pointer of an atom was contained in the address field while the CDR pointer was contained in the decrement field.

CAR returns the first element of a list. CDR returns the remaining elements of a list (which may be NIL) after the first element.

The generic formats of these two functions are

```
Function:       CAR
# Arguments:    1
Argument:       1) a nonempty list, LST
Value:          The first element of the list.
Function:       CDR
```

```
# Arguments:    1
Argument:       1) a nonempty list, LST
Value:          The remainder of the list after the
                deletion of the first element.
```

CAR always returns the first element of its argument, if it is a list. Suppose we have created the following list using SETQ:

```
←(SETQ presidents '(adams hayes monroe nixon ford))
(adams hayes monroe nixon ford)
```

When we apply CAR to this list, it returns the value

```
←(CAR presidents)
adams
```

which is the first element of the list.

CDR always returns the remainder of a list minus its first element. If we apply CDR to PRESIDENTS we obtain the result

```
←(CDR presidents)
(hayes monroe nixon ford)
```

Note that CDR always returns a list as its result if its argument is a valid list.

CAR may return either an atom or a list depending on the type of the first element in its argument. For example, if we construct the following list:

```
←(SETQ presidents-parties
        '((kennedy   democrat) (nixon republican)))
```

and then apply CAR to it, we obtain the result

```
←(CAR presidents-parties)
(kennedy democrat)
```

which is a list of two atoms. CAR returned a list because the first element of its argument was a list.

If CAR is applied to an atom, it returns an error message as a result:

```
←(CAR 'kennedy)
ARG NOT LIST
```

However, CDR applied to an atom returns NIL since we assume that the empty list is pointed to by the CDR portion of an atom.

```
←(CDR 'kennedy)
NIL
```

Both CAR and CDR may be applied to the empty list which is represented either by the atom NIL or the list '(). In this case, both functions will return the result NIL:

```
←(CAR NIL)
NIL
←(CDR '())
NIL
```

CDR operates normally on lists of one element by returning the value NIL meaning there are no more elements in the list. For example, applying CDR to the list (LINCOLN) yields

```
←(CDR '(lincoln))
NIL
```

because this list has only one element. Thus, an implied *last* element of every list is the null or empty list.

3.1.1 CAR/CDR Combinations

Lists are rarely so simple as those given above. Often, we build fairly complex structures that have multiple levels of elements. Processing these lists with CAR and CDR functions can be difficult if we need certain elements quite often.

Consider the list PRESIDENTS-PARTIES given above. How do we obtain the first element of the second sublist? One way is to take the CDR of the list, and then take its CAR:

```
←(CAR (CDR presidents-parties))
nixon
```

or

```
←(SETQ parties-temp (CDR presidents-parties))
(nixon republican)
←(CAR parties-temp)
nixon
```

Both of these methods are unwieldy. Too many CARs or CDRs used to dissect a list will result in code which is obtuse and incomprehensible. Fortunately,

INTERLISP provides us with a method of "abbreviation" for combinations of these common functions.

The general form of a combination is

```
(CxxxxR ⟨some-list-argument⟩)
```

where each 'x' represents either an A or an D. Most LISP systems will support all combinations (a total of 30) of the four-letter abbreviations.

We can rewrite the forms to retrieve the first element of the second sublist as

```
←(CADR presidents-parties)
nixon
```

We analyze such forms by reading from "right-to-left" in the function name. This form says: take the CDR of the argument, and then take the CAR of the result. Note that INTERLISP does not return the intermediate result, which is the CDR of the argument, but only the final result that we seek.

Another example is given by the following list, which is slightly more complex:

```
←(SETQ electoral-year-votes
   (LIST '(1952 eisenhower 442)
         '(1956 eisenhower 457)
         '(1960 kennedy 303)
         '(1964 johnson 486)
         '(1968 nixon 301)
         '(1972 nixon 520)))
```

which represents the election/president/electoral votes for recent elections.

Let us ask how we get the name of the winner of the election of 1956? Of course, we know the answer is EISENHOWER because we can see it in the list. But how do we dissect the list to retrieve it?

We might reason as follows:

(1956....) is the second sublist of the list.

It is also the first element of the CDR of the list so we know that we can use

```
←(CADR ⟨list⟩)
(1956 eisenhower 457)
```

EISENHOWER is the second element of a list, so following the same reasoning as above, we can produce

```
←(CADADR ⟨list⟩)
eisenhower
```

which is the desired result!

Note that we could have written this out as:

```
←(CAR (CDR (CAR (CDR ⟨list⟩))))
eisenhower
```

A few more examples should be sufficient to demonstrate the various combinations:

```
←(CADDDR electoral-year-votes)
(1964 johnson 486)

←(CDADR electoral-year-votes)
(eisenhower 457)

←(cddddr electoral-year-votes)
((1968 nixon 301) (1972 nixon 520))
```

Note that the list has too many elements to extract the last sublist using a combination form. To get the electoral votes for NIXON's second victory, we must use:

```
←(CADDDR (CDDDDR electoral-year-votes))
520
```

Here is a table of the first few combinations of CARs and CDRs that are commonly used in INTERLISP programs.

CAR/CDR Combinations

Abbreviation	**Extended Form**
(CAAR ⟨list⟩)	(CAR (CAR ⟨list⟩))
(CADR ⟨list⟩)	(CAR (CDR ⟨list⟩))
(CDAR ⟨list⟩)	(CDR (CAR ⟨list⟩))
(CDDR ⟨list⟩)	(CDR (CDR ⟨list⟩))
(CADAR ⟨list⟩)	(CAR (CDR (CAR ⟨list⟩)))
|	|
|	|
(CADDR ⟨list⟩)	(CAR (CDR (CDR ⟨list⟩)))

Here are some examples of how these abbreviations might work on the list given above:

```
←(CAAR electoral-year-votes)
1952
```

```
←(CADR electoral-year-votes)
(1956 eisenhower 457)

←(CDAR electoral-year-votes)
(eisenhower 442)

←(CDDR electoral-year-votes)
((1960 kennedy 303) .....)
```

3.2 PUTTING LISTS TOGETHER: CONS, LIST AND APPEND

In order to take lists apart, we need some way to have put them together in the first place! The three functions discussed in this section permit you to create lists in various ways. Before proceeding to a discussion of these functions, let us note that we can create lists using the SETQ function.

Remember that we said that SETQ assigns a value to an atom which is its first argument. This value may also be a list. Using SETQ, of course, the second argument is unevaluated (see Section 3.8), so the list will be typed exactly as you wish it to appear as the value of the atom. For example,

```
←(SETQ alist '(apple orange plum pear cherry lemon))
(apple orange plum pear cherry lemon)
```

sets the value of ALIST to be the specified list. If we now ask for the value of ALIST, we should receive

```
←alist
(apple orange plum pear cherry lemon)
```

We briefly mention SETQ here so that it may be used in examples in the following sections.

3.2.1 CONS: Constructing Lists

CONS is the list construction function. It takes the form

Function:	CONS
# Arguments:	2
Arguments:	1) any atom or list, EXPRESSION 2) any list, LST
Value:	A list whose CAR is the value of the first argument and whose CDR is the value of its second argument.

CONS takes two arguments, one of which is a list, and puts them together to form a new list. The first argument of CONS is always the element to be added to the second argument, which is always a list. The first element is always attached to the front of the second to form the new list. The first argument may be either an atom or a list. For example, consider the following sequence of INTERLISP statements:

```
←(SETQ subject 'john)
john
←(SETQ predicate '(is big))
(is big)
←(CONS subject predicate)
(john is big)
```

The result of the CONS is to form a *new* list having a value that is a combination of its two arguments. The first argument may also be a list. Consider the following example:

```
←(SETQ subject '(the boy))
(the boy)
←(CONS subject predicate)
((the boy) is big)
```

Note that since the first argument is a list, the first element of the new list must also be a list! This is not exactly what we wanted though. Rather, we wanted our new list to look like this:

```
(the boy is big)
```

To accomplish this, we need to dissect the first argument, and then CONS the pieces together. The following statement achieves this effect:

```
←(CONS (CAR subject)
       (CONS (CADR subject) predicate))
(the boy is big)
```

Because NIL represents the empty list, we may use it as the second argument of CONS. CONSing an atom and NIL together creates a list of a single element. Consider this example:

```
←(CONS subject NIL)
((the boy))
```

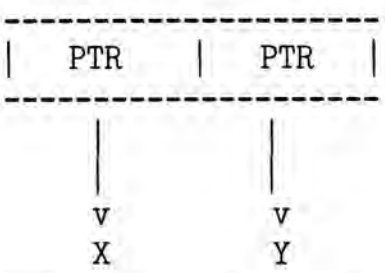
We may read this as: take the CONS of the value of "subject" and NIL, and return a list of them. This is a list of the list (the boy) and the empty list which is never represented when we print out lists unless the list value is the null list. Thus, it is possible to construct a list as follows:

```
←(CONS NIL NIL)
(NIL)
```

Because this construct is frequently used, many LISP dialects define a function, **NCONS**, which CONSes a single argument with NIL. It takes the form

```
Function:       NCONS
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          The value of the argument CONSed with NIL.
```

What happens if the second argument to CONS is not a list but an atom? CONS still executes, but it forms a data structure known as a *dotted pair*. A dotted pair is a CONS cell with pointers to two atoms—a degenerate list form. We may represent it graphically as follows:



and display it, when printing the values, as

```
(X . Y)
```

Notice the "dot" separating the two atom names whence the notion of the dotted pair.

The concept of dotted pair often causes LISP novices much trouble. I will only mention dotted pairs briefly in the remainder of this text, usually to point out where they may cause problems for users. Touretzky [tour84] and Siklossy [sikl76] both give adequate explanations of dotted pairs.

3.2.2 LIST: Making Lists

INTERLISP provides a very useful primitive function for creating a list from any number (indefinite) of arguments. This function is called **LIST**.

The format of the LIST function is shown below

```
Function:       LIST
# Arguments:    1-N
Arguments:      1-N) S-expressions, EXPRESSION[1] ...
                     EXPRESSION[N]
Value:          A list of the values of its arguments.
```

LIST takes each of its arguments in turn and places them in a list. It is a nospread function. For example, consider the following statement:

```
←(LIST 'the 'boy 'is 'big)
(the boy is big)
```

Note that the result is the same as one we previously created with multiple SETQs and CONSes. We may think of LIST in the following way:

```
(LIST ⟨argument⟩) = (CONS ⟨argument⟩ NIL)
```

Both of these statements will give the same result. The utility of the LIST function becomes apparent when we want to create much larger and more complex lists. For example, using CONS, we would have to write the following statement to create the list (the boy is big):

```
(CONS 'the
      (CONS 'boy
            (CONS 'is
                  (CONS 'big NIL))))
```

So, we may think of LIST as a shorthand notation for writing multiple CONSes to create a list. The beauty of LIST is that it take any number of arguments and gathers them up into a list. Note that it evaluates its arguments as it processes them for inclusion into the resulting list. The following example shows how we might create the list for presidential elections that we used in a previous example:

```
(LIST
     (LIST 1952 'eisenhower 442)
     (LIST 1956 'eisenhower 457)
     (LIST 1960 'kennedy 303)
     ...
```

Note that each argument to LIST becomes an element of the new list. In the example above, one argument is the S-expression (LIST 1956 'eisenhower 457).

Each of the arguments 1956, 'eisenhower, and 442 becomes an element of a list created by that LIST function. Because the form (LIST 1952 'eisenhower 442) is an S-expression, it is evaluated when it is encountered in the list of arguments to produce a value that is then included in the resultant list. Its value, of course, is (1952 eisenhower 442).

Note that LIST with no arguments returns NIL, but (LIST NIL) returns (NIL).

We said that LIST may take any S-expression as an argument. We have seen where one of those S-expressions forms a list when evaluated. Other examples of LIST to consider include

```
←(LIST '(apple orange) '(plum cherry))
((apple orange) (plum cherry))
```

Here, the arguments are lists that are passed to LIST. If the argument is already a list, LIST embeds it as a sublist within a list. For example,

```
←(LIST '(apple orange))
((apple orange))
```

But, if you do not want it to be embedded as a sublist, then you must test to see if the argument is already a list before applying LIST to it. Fortunately, INTERLISP provides a function that performs this chore for you, **MKLIST**. It takes the form

```
Function:       MKLIST
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          A list containing the value of EXPRESSION.
```

MKLIST makes a list from the value of its argument. If EXPRESSION is already a list or NIL, MKLIST merely returns the value of EXPRESSION. Otherwise, it applies LIST to the value of EXPRESSION. For example,

```
←(MKLIST 'cherry-pie)
(cherry-pie)
←(MKLIST)
NIL
←(MKLIST (LIST 1952 'eisenhower 442))
(1952 eisenhower 442)
```

We might define MKLIST as follows (although I haven't told you how to define functions yet):

```
(DEFINEQ
     (mklist (expression)
          (COND
               ((OR
                    (NULL expression)
                    (LISTP expression))
                    expression))
               (T
                    (LIST expression)))
     ))
```

3.2.3 APPEND: Concatenating Lists

A function that also creates new lists is **APPEND**. Unlike LIST, however, APPEND takes an indefinite number of lists as its arguments and returns a list by copying the lists to the new list. The generic format for invoking APPEND appears as follows

```
Function:       APPEND
# Arguments:    1-N
Arguments:      1-N) lists, LST[1] ... LST[N]
Value:          A list of the S-expressions of the
                individual lists.
```

Note that (APPEND) and (APPEND NIL NIL) both return NIL. Consider the following example:

```
←(SETQ subject '(the boy))
(the boy)
←(SETQ predicate '(is big))
(is big)
←(APPEND subject predicate)
(the boy is big)
```

Note that two individual lists have been combined into one list. Compare the two functions to see how they work:

```
←(LIST subject predicate)
((the boy) (is big))
←(APPEND subject predicate)
(the boy is big)
```

We see that after LIST is executed, the two arguments still retain their identity whereas, after APPEND is executed, they are merged into one list. We call this *top level copying* because APPEND takes the top level values and concatenates them together giving a new list.

APPEND is usually implemented as a "hardwired" primitive to provide efficient execution since it is so frequently used. We can, however, write APPEND as a recursive function using just CONS and CDR (see Chapter 8 for a discussion of function definition).

```
(DEFINEQ
     (APPEND (a-list b-list)
          (COND
               ((NULL a-list) b-list)
               (T
                    (CONS
                         (CAR a-list)
                         (APPEND (CDR a-list) b-list))))
     ))
```

but this does not copy B-LIST. It merely links the cells of B-LIST to those of A-LIST. Thus, a change to the new list created by the concatenation may also change B-LIST. It also works upon only two lists, e.g., it is a very simple definition of APPEND.

An alternative definition copies the top level elements of each argument. It is defined as follows:

```
(DEFINEQ
     (append lst
          (PROG (value x temp y z)
               (SETQ temp (CAR lst))
               (COND
                    ((AND lst
                          (NULL (CDR lst))
                          (LISTP temp))
                          (SETQ lst (CDR lst))
                          (GO loop4)))
          loop1
               (COND
                    ((NLISTP lst)
                         (RETURN value)))
               (SETQ temp (CAR lst))
               (SETQ lst (CDR lst))
               (COND
                    ((OR
```

```
                    (NLISTP lst)
                    (NLISTP temp))
                    (*
                         Only the last one of more
                         than one argument is not
                         copied.
                    )
                    (GO loop2)))
          loop4
               (SETQ z
                    (SETQ y
                         (CONS (CAR temp)
                              (CDR temp))))
          loop3
               (SETQ temp (CDR temp))
               (COND
                   ((LISTP temp)
                        (SETQ y
                             (RPLACD y
                                  (CONS (CAR temp)
                                       (CDR temp))))
                        (GO loop3)))
               (SETQ temp z)
          loop2
               (COND
                   ((LISTP x)
                        (RPLACD (SETQ x (LAST x))
                                temp))
                   (T
                        (SETQ value (SETQ x temp))))
               (GO loop1))
     ))
```

APPEND will also work on arguments that are not lists. However, the data structures that are produced involved dotted pairs. Consider the following examples:

```
←(APPEND 'kennedy
         (LIST 'johnson 'nixon 'ford 'carter))
(johnson nixon ford carter)
←(APPEND (LIST 'johnson 'nixon 'ford 'carter)
         'reagan)
(johnson nixon ford carter . reagan)
```

```
←(APPEND
    (APPEND (LIST 'johnson 'nixon 'ford)
            'carter)
    (LIST 'reagan))
(johnson nixon ford reagan)

←(APPEND
    (APPEND (LIST 'johnson 'nixon 'ford)
            'carter))
(johnson nixon ford . carter)
```

Note that APPEND is modifying the values of the CDR pointers in the CONS cells when it combines lists.

3.2.4 Creating (NIL)

Sometimes, we will find it useful to create a list containing the NIL list. This list takes the form (NIL). We can create this list in two simple but elegant ways using the primitive functions CONS and LIST.

With CONS, we merely say

```
←(CONS)
(NIL)
```

which returns (NIL) because both of its argument are NIL.

With LIST, we can merely say

```
←(LIST NIL)
(NIL)
```

where the second argument is an implied NIL that is treated as the empty list because that is what LIST expects.

Note that APPEND cannot be used to return (NIL) because it operates upon lists. Thus

```
←(APPEND)
NIL

←(APPEND NIL NIL)
NIL
```

3.3 PHYSICAL STRUCTURE REPLACEMENT: RPLACA AND RPLACD

In Section 2.4, we discussed the structural representation of lists using cells. This provided a convenient way of thinking about lists as collections of pointers

to values as well as boxes for holding values. Using this model, whenever we created a result from a function so far, we have always allocated new cells to hold the result. This approach can be quite expensive, both in terms of execution efficiency and in consumption of free memory.

INTERLISP provides us with two functions that allow us to physically modify a list without allocating new cells. These functions, RPLACA and RPLACD, allow us to replace the contents of the CAR or the CDR portions of a list cell, respectively.

3.3.1 Replacing the CAR Cell

RPLACA (RePLAce CAr) allows us to replace the CAR portion of a list cell. RPLACA takes the following form

```
Function:       RPLACA
# Arguments:    2
Arguments:      1) an S-expression resolving to an atom,
                ATM
                2) an S-expression, EXPRESSION
Value:          The new value of the literal atom after
                the replacement has been performed.
```

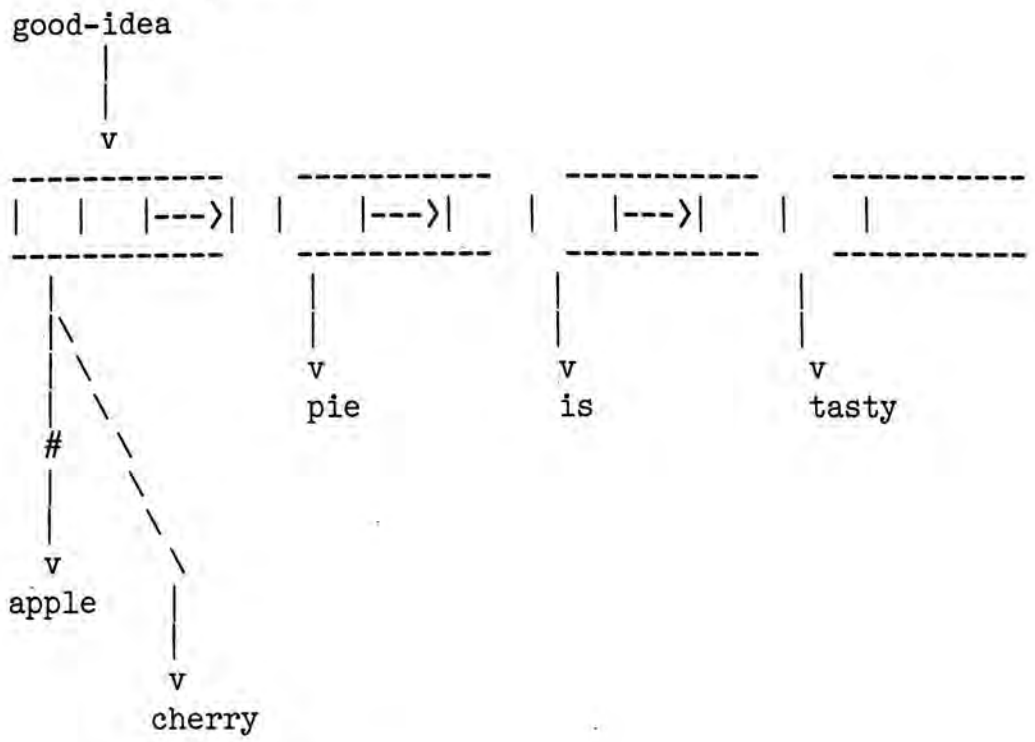
Suppose we have created a list of the following form:

```
←(SETQ good-idea '(apple pie is tasty))
(apple pie is tasty)
```

We can change the type of pie by using RPLACA to change the CAR portion of the first list cell (e.g., the one containing "apple") as follows:

```
←(RPLACA good-idea 'cherry)
(cherry pie is tasty)
```

Note that INTERLISP allocates no new cells to hold the result but merely changes the pointer to the atom APPLE to the pointer to the atom CHERRY. We say that RPLACA "smashes" the CAR value of the CONS cell. We can diagram this as follows:



where # indicates that the pointer to APPLE has been broken (e.g., no longer exists).

Errors may occur when we give RPLACA bad arguments. For example, we cannot RPLACA NIL. Consider the statement

```
←(RPLACA NIL ⟨anything⟩)
ATTEMPT TO RPLAC NIL
```

except for the form (RPLACA NIL NIL) which has no effect at all:

```
←(RPLACA NIL NIL)
NIL
```

If we attempt to replace the CAR portion of something that is not a list, INTERLISP returns the error ARGUMENT NOT LIST. Consider the statement

```
←(RPLACA 'apple ⟨anything⟩)
ARGUMENT NOT LIST
```

3.3.2 Replacing the CDR Cell

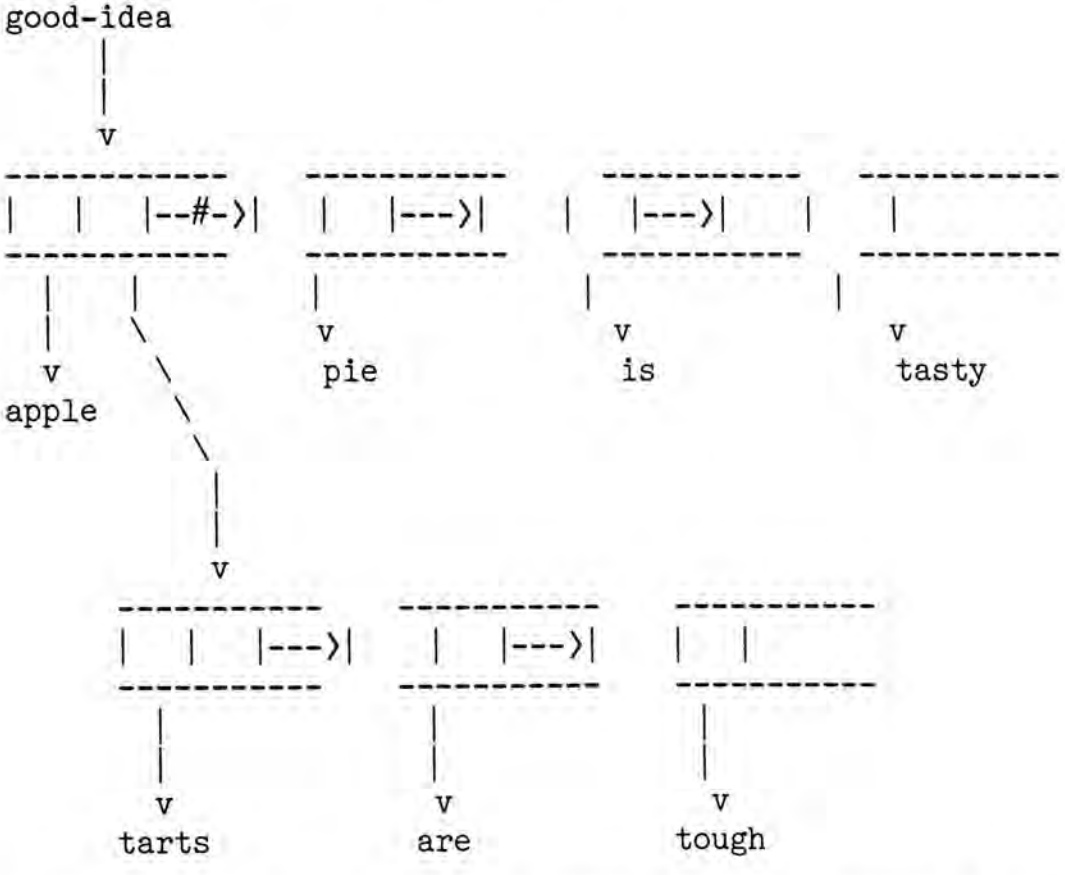
In a similar fashion, **RPLACD** (RePLAce CDr) replaces the pointer to the CDR portion of a list cell. RPLACD takes the following form:

```
Function:       RPLACD
# Arguments:    2
Arguments:      1) an S-expression evaluating to a literal
                atom, ATM
                2) an S-expression, EXPRESSION
Value:          The new value of the literal atom after
                the replacement has been performed.
```

Using the list GOOD-IDEA defined above, we can use RPLACD to create a list that has as its value (apple tarts are tough).

```
←(RPLACD good-idea '(tarts are tough))
(apple tarts are tough)
```

The following diagram shows how this is done:



where # indicates that the pointer from the CDR portion of the CONS is broken.

Using RPLACD can be dangerous for a number of reasons. If you change the pointer to a group a cells to which nothing else points, those cells are lost. For example, the cells comprising the list (PIE IS TASTY) are pointed to by the

CDR portion of GOOD-IDEA By replacing that pointer, we have no knowledge of the address of the first cell of that list. Thus, the cells are allocated, but we have no way to access them again during the remainder of the INTERLISP session. Too many mistakes in this fashion causes memory to be consumed at a prodigious rate. One indication that memory is disappearing is a more frequent occurrence of garbage collections.

Another problem that you may encounter is creating circular lists. Circular lists may be good or bad, depending on your application's requirements. In most cases they are bad. We can create a circular list by causing the CDR portion of a list cell to point to the beginning of the list. For example, consider the following statement:

```
←(RPLACD (CDDDR good-idea) good-idea)
... an infinite list ...
```

causes the CDR portion of the last cell of GOOD-IDEA to point to the first cell of the list.

Errors may occur if we give RPLACD bad arguments. For example, we cannot replace the CDR portion of NIL:

```
←(RPLACD NIL ⟨anything⟩)
ATTEMPT TO RPLAC NIL
```

except that INTERLISP allows us to say (RPLACD NIL NIL) which has no effect at all:

```
←(RPLACD NIL NIL)
NIL
```

If we attempt to RPLACD something that is not a list, INTERLISP returns the error ARGUMENT NOT LIST. For example, consider

```
←(RPLACD 'apple ⟨anything⟩)
ARGUMENT NOT LIST
```

3.3.3 Replacing the CAR and CDR of a Cell

INTERLISP provides two functions to replace the contents of a node (e.g., an atom) without changing the atom's name: RPLNODE and RPLNODE2. **RPLNODE** takes the form

```
Function:       RPLNODE
# Arguments:    3
```

```
Arguments:      1) an atom, ATM
                2) an S-expression, EXPRESSION-A
                3) an S-expression, EXPRESSION-D

Value:          The name of the atom.
```

RPLNODE (RePLace NODE) replaces the CAR and CDR pointers of the atom's CONS cell without changing the name of the atom or creating a new CONS cell.

We might define RPLNODE as follows:

```
(DEFINEQ
     (rplnode (atm expression-a expression-d)
          (RPLACA atm expression-a)
          (RPLACD atm expression-d)
          atm
     ))
```

RPLNODE2 also replaces the contents of an atom's CONS cell by extracting the CAR and CDR portions of an S-expression. It takes the form

```
Function:       RPLNODE2

# Arguments:    2

Arguments:      1) an atom, ATM
                2) an S-expression, EXPRESSION

Value:          The name of the atom.
```

We might define RPLNODE2 as follows:

```
(DEFINEQ
     (rpnode2 (atm expression)
          (RPLACA atm (CAR expression))
          (RPLACD atm (CDR expression))
          atm
     ))
```

3.4 PREVENTING EVALUATION

In programming languages, we need to differentiate between a symbol that stands for something (i.e., a variable) and a literal (i.e., a symbol whose value is itself). Numbers are literals. Atoms are variables.

When we use atoms in INTERLISP statements, we assume the atom has a value. What we want to manipulate is the value of the variable, not the name of the variable. For example,

```
(IPLUS x y)
```

means add the value of X to the value of Y and return the result.

Arguments are usually passed to INTERLISP functions using the call-by-value method. That is, what the function "sees" is not the name of the variable, but its value. So, in evaluating the statement above, what IPLUS sees are the values of the variables X and Y. INTERLISP evaluates the two arguments to determine their values and passes these values to the function.

Sometimes we do not want arguments evaluated before they are passed to a function. In Chapter 8, we shall see one method of preventing evaluation in the way in which certain types of functions (called NLAMBDA functions) are defined. Another way is to QUOTE the argument so that its literal value is passed rather than attempting to evaluate it for a value.

Suppose we want to test the equality of two values. First, we may represent them as the values of atoms. For example,

```
←(SETQ x 'apples)
apples
←(SETQ y 'oranges)
oranges
←(EQUAL x y)
NIL
```

because the two values are not equal. We could directly specify the value ORANGES instead of assigning it to a variable as follows:

```
←(EQUAL x 'oranges)
NIL
```

If we had attempted to execute

```
←(EQUAL x oranges)
U.B.A.
oranges
```

because INTERLISP expects ORANGES to be the name of a variable that has a value. By *quoting* ORANGES, we tell INTERLISP that it should pass the literal value ORANGES to the EQUAL function.

Some functions in INTERLISP automatically assume that one or more of their arguments are not to be evaluated; that is, they ar assumed to be "quoted." SETQ is one of these. It assumes that its first argument (for example, X) is to be assigned a value. A variant of SETQ which assumes that both its arguments are quoted is SETQQ. Compare the following examples:

```
←(SETQ x1 'apples)
apples

←(SETQQ x2 apples)
apples

←x1
apples

←x2
apples
```

The INTERLISP function that prevents the evaluation of its argument is **QUOTE**. It takes the form

```
Function:        QUOTE
                 KWOTE

# Arguments:     1

Argument:        1) an S-expression, EXPRESSION

Value:           The literal value of its argument.
```

QUOTE is an NLAMBDA, nospread function. QUOTE returns the PRIN1-PNAME of that argument. However, because we use the quoting facility so much in INTERLISP, a convenient shorthand notation is provided for entering the QUOTE function. We prefix the S-expression to be quoted by a single apostrophe (sometimes called a *quote-mark*). The single apostrophe is treated as a read macro (see Section 14.5).

Internally, INTERLISP requires all data to be expressed in a strict S-expression form. The quote-mark notation does not fit this form. So, the INTERLISP function that reads data typed in by the user converts all '⟨S-expression⟩ forms to (QUOTE ⟨S-expression⟩) forms. Thus, while we type in

```
(CAR 'x)
```

what INTERLISP really sees is

```
(CAR (QUOTE x))
```

When we attempt to give QUOTE more than one argument, it generates a PARENTHESES ERROR. For example,

```
←(QUOTE i (CONS see her))
PARENTHESES ERROR
```

because QUOTE does not know how to process the second argument.

An alternate function, **KWOTE**, returns a value which is its argument as a literal value. KWOTE is a LAMBDA function; that is, it arguments are evaluated. For example,

```
←(SETQ x 'apples)
apples
←(SETQ y 'oranges)
oranges
←(KWOTE (list x y))
(QUOTE (apples oranges))
```

If the value of its argument is a number, which is a literal value, KWOTE merely returns that number:

```
←(SETQ pi 3.141592)
3.141592
←(KWOTE pi)
3.141592
```

Similarly, if the value of the argument is NIL, KWOTE merely returns NIL:

```
←(SETQ truth-flag NIL)
NIL
←(KWOTE truth-flag)
NIL
```

3.5 CONDITIONAL EXECUTION: COND

INTERLISP provides a conditional execution facility through the function **COND**. The format of the COND expression is

Function:	COND
# Arguments:	1-N
Arguments:	any S-expressions
Value:	The value of the last statement in the first S-expression whose CAR evaluates to T.

COND is an NLAMBDA, nospread function. A COND expression takes the following format (when prettyprinted):

```
(COND
     (⟨test1⟩ ... ⟨result1⟩)
     (⟨test2⟩ ... ⟨result2⟩)
         .            .
         .            .
     (⟨testN⟩ ... ⟨resultN⟩))
```

3.5.1 Executing a COND Expression

The elements of a COND statement are called *clauses*. Each clause is composed of a test phrase, represented as an S-expression, followed by zero or more action phrases, represented as S-expressions, which are actions to be taken if the test evaluates true.

CONDs are executed as follows:

Execute the test of the first clause. If it is true (i.e., T), then execute all of the S-expressions comprising the result. The value of the COND is the value of the last S-expression in the result that is executed.

If the test returns NIL when evaluated, proceed to the next clause.

Continue to evaluate the tests of each clause in turn until one returns a non-NIL value. If this occurs, execute the S-expressions in the result as mentioned above.

If no clause evaluates successfully, the value of the COND is NIL.

Suppose we wanted to determine the type of an argument presented to a function. We could use COND to execute a sequence of tests that could serve to identify the argument's type. A possible definition for a function that captures this idea is

```
(DEFINEQ
     (test-argument (an-argument)
          (COND
               ((LISTP an-argument) 'LIST)
               ((NUMBERP an-argument) 'NUMBER)
               ((STRINGP an-argument) 'STRING)
               ((ATOM an-argument) 'ATOM)
               (T 'UNKNOWN))
     ))
```

In this example, there are five clauses. The first clause consists of the test

```
(LISTP an-argument)
```

and the result

```
'LIST
```

If the argument passed to TEST-ARGUMENT were a list such as (A B), then TEST-ARGUMENT would return the value LIST. LISTP is a predicate (see Chapter 4) which returns either a true (T) or false (NIL) value.

3.5.2 The Default Clause

The last clause (T NIL) in the example above is often called a *default clause* because it will be executed if no other clause succeeds. The test in the default clause always evaluates to true because it is the single atom T. Thus, the result will always be executed. For example,

```
←(SETQ x (ARRAY 10 5 5))
[ARRAYP]#1,1044
←(test-argument x)
unknown
```

because X is not one of the data types tested for in TEST-ARGUMENT. Thus, none of the first four clauses succeeds. The default clause is executed and returns UNKNOWN.

Because a clause with T as its test is always evaluated, it should always be the last clause in the COND expression. Clauses appearing after the T clause in a COND expression will never be executed because they can never be reached during normal program flow. This is a frequent error made by novice LISP programmers that is easy to avoid.

3.5.3 Test Phrase Values

The test of a clause must return a true or false value to determine whether the result is executed or not. Many functions do not return T but some value (for example, MEMBER). If we required the value of the test to be only T or NIL, then any function returning other than T would have to be tested for a non-NIL value to produce a T. Consider the following COND fragment:

```
(COND
     ((MEMBER item bag) item)
     ....
```

MEMBER always returns the fragment of a list beginning with its first argu-

ment if that first argument is indeed a member of the list (see Section 4.8). Thus, to ensure a T or NIL value, we would have to test what MEMBER returns via

```
←(NULL (MEMBER item bag))
T or NIL
```

Happily, INTERLISP does not require this constraint. Any value returned by a function used in the test phrase of a clause that is non-NIL suffices to indicate a successful execution. Thus, if (MEMBER item bag) returns a non-NIL value, the corresponding result phrases will be executed.

Suppose we have a program which reads user queries and executes them against a database. In addition, the program may also respond to several commands that cause other functions to be called to perform administrative duties associated with program execution. We might write the function to read the query and decide what to do as follows (making judicious use of the COND expression):

```
(DEFINEQ
     (get-query (query)
          (PROG (first-character)
               (COND
                    ((NOT (NULL query))
                         (RETURN query )))
          loop
               (PRIN1 ''Enter Query: '')
               (SETQ query (READ))
               (SETQ first-character
                    (CAR (UNPACK query)))
               (COND
                    ((EQUAL (CAR (CHCON first-character))
                            (CAR (CHCON '#)))
                            (SETQ query
                                 (PACK
                                    (CDR
                                       (unpack query))))
                            (execute-query query))
                    ((EQUAL (CAR (CHCON first-character))
                            (CAR (CHCON '?)))
                            (print-commands))
                    ((EQUAL 'DEBUG query)
                            (SETQ *debug* (NOT *debug*)))
                    ((EQUAL 'OPTIONS query)
                            (set-options))
                    ((EQUAL 'RULES query)
                            (show-rules))
```

```
                    ((EQUAL 'QUIT query)
                            (RESET))
                    (T
                        (RETURN query)))
                (GO loop))
))
```

3.6 MULTIPLE CASE SELECTION: SELECTQ

The COND statement tests each clause until it finds one that returns a true value, executes its consequents, and exits. Often, you will want to perform some function based on an explicit value of a variable or function. In a COND statement, you would have to test for each and every instance, perhaps through using EQUAL. This can be both tedious as well as leading to omitting one or more important cases. INTERLISP provides the **SELECTQ** function to select a specific statement from among multiple cases.

A SELECTQ expression takes the form

```
Function:       SELECTQ

# Arguments:    2-N

Arguments:      1) a selection phrase, SELECTOR
                2) a case clause, CLAUSE[1]
                3-N) case clauses, CLAUSE[2] ... clause[N]

Value:          The value of the last S-expression
                executed within the selected case clause
                or the default clause.
```

SELECTQ is an NLAMBDA, nospread function. The general structure of the SELECTQ statement is given by:

```
(SELECTQ ⟨selection-value⟩
     (case[1]    ⟨clause[1]⟩)
     (case[2]    ⟨clause[2]⟩)
        .           .
        .           .
     (case[N]    ⟨clause[N]⟩)
     (⟨default-clause⟩))
```

3.6.1 Executing a SELECTQ Expression

The ⟨selection-value⟩, SELECTOR, may be an atom or an S-expression that evaluates to an atomic value. It is compared with the case clauses as follows:

1. If case[i] is an atom, then if (EQ SELECTOR case[i]) is true, INTERLISP executes the S-expression comprising clause[i].

2. If case[i] is a list, then SELECTOR is compared with each element of the list in turn. If it is EQ to one of the elements of the list, then the S-expressions in clause[i] are executed.
3. If none of the case[i] are selected by either of the two methods, the ⟨default-clause⟩ is executed. There must always be a default clause present in a SELECTQ expression.

Note that none of the case[i] are evaluated. Moreover, the clause[i] may be compound expressions as denoted by a PROGN (see Section 3.7.2).

The value of a SELECTQ statement is always the value of the last S-expression executed in any of the clauses or the ⟨default-clause⟩.

3.6.2 SELECTQ Examples

Suppose we can select one of several operations from a menu. How might we use SELECTQ to "switch" to the proper execution stream for the selected operation? Let us assume that the operations are CREATE, DELETE, UPDATE, DISPLAY, and EXIT. We might define a function as follows:

```
(DEFINEQ
     (execute.operation (item menu key)
          (COND
               ((EQUAL key 'MIDDLE)
                    (SELECTQ (CAR item)
                         (create
                              (create.node))
                         (update
                              (update.node))
                         (delete
                              (delete.node))
                         (display
                              (display.node))
                         (exit
                              (exit.operations))
                         (PROGN NIL))))
     ))
```

This example is drawn from a program in INTERLISP-D that makes use of the menu display and selection capability in the window system. You select an item from the menu by pressing the mouse key. Associated with the menu is a function to be executed when a selection is made. The function above "switches" you to the appropriate function to be executed based on your selection. Note that the last expression in the SELECTQ statement is a PROGN which means do nothing if no selection was made.

The IRM [irm83] presents another simple example to select the number of days in a month. It looks like this:

```
(SELECTQ month
     (february
          (COND
               ((leapyearp)
                    29)
               (T 28)))
     ((april june september november) 30)
     (PROGN 31))
```

where LEAPYEARP tests if the current year is a leap year (e.g., divisible by 4).

SELECTQ may also be used to invoke different processing routines based on the current state of the computation. Consider the case where your program is reading free text from a file. Your program must perform different kinds of processing based on the syntactic element it is trying to complete. Here is a fragment of a routine showing how SELECTQ might be used:

```
(SELECTQ state
     (scanning
          (COND
               ((test-character (LIST   *PARAGRAPH*
                                        *DELIMITER*
                                        *SENTENCE*)))
               (T
                    (SETQ word (TCONC NIL char))
                    (SETQ state 'word))))
     (word
          (COND
               ((test-character *DELIMITER*)
                    (PACK word))
               (T
                    (TCONC word char))))
     (sentence
          (COND
               ((test-character *NEW-LINE*)
                    '*SENTENCE*)
               (T
                    (SETQ state 'scanning))))
     (paragraph
          (COND
               ((test-character *NEW-LINE*)
                    '*PARAGRAPH*)
               (T
                    (SETQ state 'scanning)))))
```

The basic idea is that STATE holds the current state of the scanning process. In this example, SELECTQ acts like a finite state automaton to switch the computation to the current processing routine based on the input it has just received.

3.6.3 A Definition for SELECTQ

We might define SELECTQ as follows:

```
(DEFINEQ
     (selectq
          (NLAMBDA selectq-args
               (APPLY 'PROGN
                    (SELECTQ1
                         (EVAL (CAR selectq-args))
                         (CDR selectq-args)))
     )))
```

Note that we must evaluate the SELECTOR to obtain its value for comparison with the cases.

SELECTQ1 is defined as

```
(DEFINEQ
     (selectq1 (selector clauses)
          (PROG (clause-list)
               (SETQ clause-list clauses)
          loop
               (SETQ clauses (CDR clauses))
               (COND
                   ((NULL clauses)
                        (*
                             A single case/clause pair
                             in the SELECTQ, so return
                             the sole clause.
                        )
                        (RETURN clause-list))
                   ((OR
                        (EQ
                             (CAR (SETQ clause-list
                                        (CAR clause-
                             list)))
                             selector)
                        (AND
                             (LISTP (CAR clause-list))
                             (MEMBER selector
                                     (CAR clause-
                             list))))
```

```
                                (*
                                    The first of these
                                    expressions tests the
                                    selector against a single
                                    atom in a case.
                                    The second expression tests
                                    against a list of atoms in
                                    the case.
                                )
                                (RETURN (CDR clause-list))))
                        (GO loop))
        ))
```

Note that SELECTQ1 merely returns a list of the clauses to be executed when a match has been found for a particular case. The clauses are executed in the APPLY expression in SELECTQ through the application of PROGN.

3.6.4 SELECTC: Selecting on Constants

A variation on SELECTQ is the function **SELECTC** which performs selection on constants. It takes the form

Function:	SELECTC
# Arguments:	2-N
Arguments:	1) a selection phrase, SELECTOR 2) a case clause, CLAUSE[1] 3-N) optional case clauses, CLAUSE[2] ... CLAUSE[N]
Value:	The value of the last expression executed in the case clause that is selected.

SELECTC is an NLAMBDA, nospread function. SELECTC allows you to determine the keys in the case phrase of a case clause at execution time. In SELECTQ, the key(s) in a case are literals that are not evaluated at execution time. However, the case[i] which determine the selection keys may be S-expressions which are evaluated to produce the possible selection keys. SELECTC is compiled as a SELECTQ, so that its selection keys are treated as compile-time constants.

The IRM [irm83] gives an example of how SELECTC may be used:

```
(SELECTC number
    ((for X from 1 to 9
          collect (TIMES X X))
          "SQUARE")
    (PROGN "NON-SQUARE"))
```

where the (for ...) expression is evaluated at execution time to produce the list (1 4 9 16 25 36 49 64 81) against which the selector (e.g., NUMBER) is compared.

3.7 ITERATIVE EVALUATION: PROG

The iteration mechanism provided by INTERLISP is the **PROG** statement. PROG allows you to write a "program" that may transfer control either forward ("skipping") or backward ("looping") over one or more statements.

A PROG expression takes the format

```
Function:       PROG
# Arguments:    1-N
Arguments:      1) a list of variables, VARLST
                2-N) one or more S-expressions,
                EXPRESSION[1] ... EXPRESSION[N]
Value:          The value of the last S-expression
                executed within the scope of the PROG.
```

PROG is an NLAMBDA, nospread function. The general structure of a PROG statement is

```
(PROG     ⟨program-variable-list⟩
                .
          ⟨S-expressions⟩
                .
⟨label⟩
                .
          ⟨S-expressions⟩
                .
          (GO ⟨label⟩))
```

PROG operates as follows. The program variables specified in the ⟨program-variable-list⟩ are initialized to NIL or to a specified value (see below). The ⟨S-expressions⟩ are executed in sequence. Control of statement execution may be modified in two ways:

1. A statement of the form (GO ⟨label⟩) is executed that specifies the next statement to be executed is found after ⟨label⟩. Control may be transferred either forward or backward within the PROG body.
2. A statement of the form (RETURN ⟨S-expression⟩) is executed which causes the PROG to immediately exit with the value of the S-expression.

Control may be transferred either forward or backward within the list of S-expressions depending on the position of the ⟨label⟩ that is the argument of GO.

Labels must always be literal atoms. They serve only as markers within the sequence of S-expressions and are never executed.

The value of a PROG statement is either the value of the RETURN statement or NIL if the PROG "falls off the end." That is, the last S-expression in the sequence is executed without a RETURN statement being encountered. The latter form is bad programming practice.

3.7.1 Binding of PROG Variables

A ⟨program-variable-list⟩ specifies the variables that are used by the PROG statement. They are similar to LAMBDA variables (see Chapter 8) in that they are bound locally to the PROG expression. Once the PROG expression is executed, the PROG variables cease to have a valid binding. If no program variables are needed, you must specify NIL or () to indicate no local variables are needed. Otherwise, ⟨program-variable-list⟩ entries may take one of two forms:

1. An entry may be a literal atom which is then initialized to NIL. For example,

```
(PROG (clause) ...)
```

2. An entry may have the form (⟨atom⟩ ⟨S-expression⟩) where the atom is initialized to the value determined by evaluating the S-expression. For example,

```
(PROG ((sum 0) (index 1)) ...)
```

Attempting to use anything other than a literal atom as a PROG variable causes the error message ARG NOT LITATOM to be printed. You may not use NIL or T as PROG variable names, although they may be used to initialize PROG variables. Attempting to do so will cause the error message ATTEMPT TO BIND NIL OR T to be displayed and an error to occur.

PROG variables exist only for the execution of the PROG. They have no value outside it. Thus, once you execute a RETURN statement, any PROG variables within the PROG disappear (i.e., become undefined).

PROG variables do not have to be unique. You may use the names of variables that are external to the PROG as the names of PROG variables. However, atom names appearing in a ⟨program-variable-list⟩ take precedence over those names external to the PROG. That is, the name indicates a new variable which has the value given by the PROG initialization and not the value of a similarly named variable external to the PROG.

Here is an example of a PROG expression used in a function that skips spaces while reading text from a file.

```
(DEFINEQ
    (skip-spaces (file-in)
        (PROG (char)
        loop
              (SETQ char (read-character file-in))
              (COND
                  ((NULL char)
                       (RETURN *EOF*))
                  ((EQUAL
                       (CAR (CHCON char))
                       *EOF*)
                       (RETURN *EOF*))
                  ((NOT (EQUAL
                            (CAR (CHCON char))
                            space))
                            (RETURN char))
                  (T   (GO loop))))
    ))
```

Note that the RETURN expressions are embedded within the COND clauses. We could just as easily have written this function as follows:

```
(DEFINEQ
    (skip-spaces (file-in)
        (PROG (char)
        loop
              (SETQ char (read-character file-in))
              (COND
                  ((NULL char)
                       (GO end-of-file))
                  ((EQUAL
                       (CAR (CHCON char))
                       *EOF*)
                       (GO end-of-file))
                  ((NOT (EQUAL
                            (CAR (CHCON char))
                            space))
                            (GO exit))
                  (T   (GO loop)))
        end-of-file
              (RETURN *EOF*)
        exit
              (RETURN char))
    ))
```

This form makes the transfer of control more explicit in that the function skips forward to labels to exit the program. The style is akin to FORTRAN programming.

3.7.2 Variations on PROG

There are three variations to PROG. **PROG1** takes a sequence of S-expressions with no ⟨program-variable-list⟩, executes each in turn, but always returns the value of the first S-expression that it executed. **PROG2** is similar to PROG1 but returns the value of its second argument. **PROGN** is a function which evaluates its arguments in order and returns the value of the last argument. All are NLAMBDA, nospread functions. They take the form

Function:	PROG1 PROG2 PROGN
# Arguments:	1-N
Arguments:	1-N) S-expressions
Value:	The value of the first (respectively, the second) S-expression in the argument list.

Suppose we had a function that we knew was used to read the first atom of a command line. When it is called, it reads the atom, via RATOM, and also sets up the margin for echoing the command line. We want this function to return the atom read even though we test for the existence of a command and perform an additional function. *RATOM can do this using PROG1:

```
(DEFINEQ
     ( *ratom ()
          (PROG1
               (SETQ command (RATOM))
               (AND
                    (IS.COMMAND? command)
                    (MAKEMARGIN)))
     ))
```

Note that I have defined *RATOM with a space between the (and the function name *RATOM so that it is distinguished from a comment.

PROGN evaluates each of the S-expressions which are its arguments, but always returns the value of the last S-expression.

PROG1, PROG2, and PROGN may be thought of as block delimiting statements. That is, they identify a sequence of statements that are executed as a single entity. Their only difference is which S-expression value they return. Pro-

grammers familiar with such block-structured languages as C and PASCAL will see a striking similarity to the BEGIN ... END blocks of those languages.

PROG1 and PROGN are particularly useful in SELECTQ expressions. The cases of a SELECTQ expression take the form

```
(⟨selector⟩    ⟨action⟩)
```

The syntax of SELECTQ restricts ⟨action⟩ to a single S-expression. Using PROGN, we may collect any number of S-expressions into a single expression. For example,

```
(⟨selector⟩
     (PROGN    ⟨action1⟩
               ⟨action2⟩
                    .
                    .
               ⟨actionN⟩))
```

Note also that RETURN (see Section 3.7.4) takes a single S-expression as its argument. PROGN may be used here as well to execute a block of S-expressions the last of which becomes the value of the PROG.

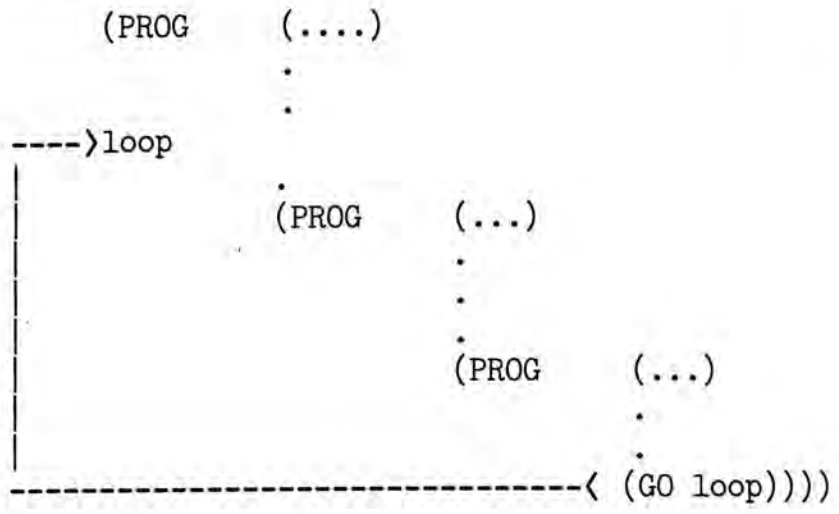
3.7.3 Transfer of Control

PROG expressions allow you to develop iterative procedures in a function. Control is transferred to another statement by executing **GO**. It takes the form

Function:	GO
# Arguments:	1
Arguments:	1) a label, LABEL
Value:	None, but control is transferred to LABEL, which must be a literal atom.

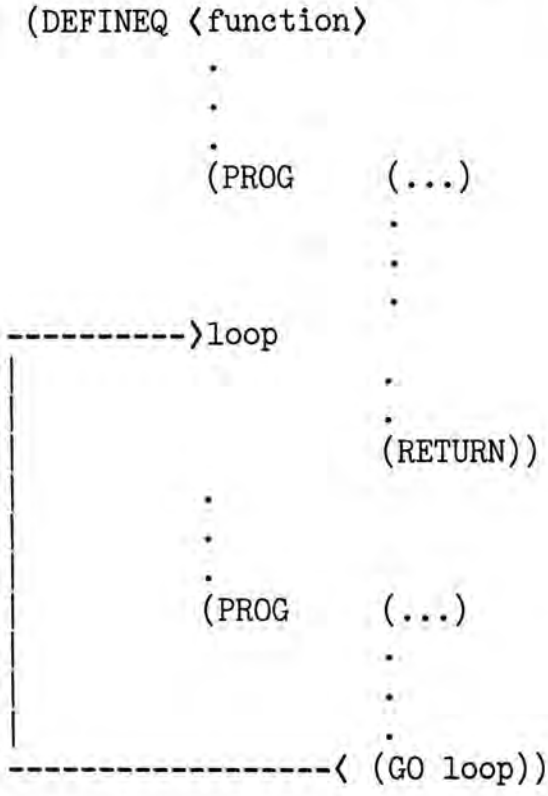
LABEL identifies a location within the body of the PROG. If the label is undefined, GO generates an error with the message "UNDEFINED OR ILLEGAL GO".

GO transfers control only within a function in which the PROG is defined. GO may transfer control either forward or backward within a PROG expression. PROG expressions may be nested within one another to any depth. When GO is executed, if the ⟨label⟩ does not appear within the current PROG, INTERLISP searches the hierarchy of PROGs looking for ⟨label⟩. Control is transferred to a statement higher in the hierarchy if it is identified by ⟨label⟩. For example,



The GO expression transfers control to the label LOOP in the outermost PROG.

Sometimes, PROGs may be independently situated within a function. Control may not be transferred out of one PROG and into another because, once a PROG has been exited, no knowledge exists about its structure. For example,



will generate an error because no knowledge of LOOP is retained once the first PROG has been exited.

Control may be transferred either forward or backward within a PROG body. As in conventional programming languages, there are many possibilities for abusing the unconditional GOTO.

I recommend limited use of it according to the following rules:

1. GO should transfer control only backward within a PROG body, except for rule 2;
2. O may transfer control forward only to a label that identifies the last statement in the PROG body. For mnemonic purposes, this label should have the word EXIT as part of its name to indicate that the statements that follow will terminate the PROG.

3.7.4 Exiting PROGs

PROG expressions may be exited in two ways:

1. Normally, by executing a **RETURN** expression.
2. Abnormally, by "dropping off" the end of the PROG body.

When a PROG expression terminates, it normally returns NIL as its value. RETURN allows us to stipulate a value that we want returned as the value of the PROG expression.

RETURN takes the form

`Function:`	`RETURN`
`# Arguments:`	`1`
`Argument:`	`1) an S-expression, EXPRESSION`
`Value:`	`The value of the S-expression, but a side effect is to terminate the PROG expression.`

EXPRESSION is an expression which is evaluated and becomes the value of the PROG.

If RETURN is executed inside an interpreted function, but not within a PROG, it will force an exit from the last interpreted PROG expression that was entered, if any. Otherwise, an error will result. That is, you may call other functions within the body of a PROG expression and embed the RETURN from the PROG within one of those functions. However, this can lead to considerable confusion when reading program code and, I believe, constitutes poor programming practice.

The compiler detects RETURN expressions that are not contained within PROG expressions in a function and generates an error at compile time.

3.7.5 Implementing a DO-WHILE-UNTIL Construct

Charniak et al. [char80] describe the implementation of a LOOP macro (using MACLISP features) that includes both DO-WHILE and DO-UNTIL capabili-

ties. INTERLISP does not provide macro features at the user programming level. Nevertheless, let us describe how the LOOP statement works as a guide for building PROG statements.

The basic structure of the LOOP statement consists of a number of subordinate statements organized in sequence:

```
(LOOP
     (INITIAL ⟨initialization-expressions⟩)
     (WHILE   ⟨while-condition⟩)
     (DO      ⟨do-body⟩)
     (NEXT    ⟨next-case-expressions⟩)
     (UNTIL   ⟨until-condition⟩)
     (RESULT  ⟨return-condition⟩))
```

Most of these subordinate statements are optional. They may be combined in several ways to provide analogues to conventional DO-WHILE or DO-UNTIL iterations. Indeed, you may specify an INITIAL-DO-RESULT loop which acts exactly like a PROGN.

Each of the subordinate statements is translated into a more familiar LISP statement by the macro. We will examine each of the statements and then show how they map into a general construct for an INTERLISP PROG statement.

1. The INITIAL statement specifies the local variables of the PROG and assigns them values prior to the execution of any statements in the loop body. The format of an ⟨initialization-expression⟩ is a sequence of ⟨variables-expression⟩ pairs. For example,

   ```
   (INITIAL SUM 0 COUNT 0)
   ```

   initializes two variables, SUM and COUNT, to 0. However, the expression may be any valid S-expression.

   The INITIAL statement is translated into a sequence of SETQ statements plus a list of variables that become the ⟨program-variable-list⟩ of the PROG statement. Thus, the example above becomes

   ```
   (PROG (SUM COUNT)
         (SETQ SUM 0)
         (SETQ COUNT 0)
   ```

2. The WHILE statement tests a condition and terminates the loop if the condition yeilds a false (NIL) value. Stated in a different way, the loop body is executed as long as the ⟨while-condition⟩ is true (T). The ⟨while-condition⟩ is just an S-expression that is to be evaluated. Thus, the WHILE statement translates as

```
(OR
     (⟨while-condition⟩)
     (GO EXIT$))
```

where EXIT$ is a label indicating the exit from the loop. Note that Charniak's macro generator inserts this automatically, but in our formulation you will have to code it explicitly. As an example, consider

```
(WHILE (NEQ COUNT 10))
```

which translates to

```
(OR
     (NEQ COUNT 10)
     (GO EXIT$))
```

3. The DO statement specifies the body of the loop. It contains one or more S-expressions that are to be executed on each pass through the loop. The expressions are evaluated from first to last in order. Transfers of control occurring within the DO-body must be explicitly encoded by the user. An example of a DO-body is

```
(DO
     (SETQ NEXT-NUMBER (READ))
     (SETQ SUM (IPLUS SUM NEXT-NUMBER)))
```

which just reads a number and adds it to SUM. The translation of this DO-body is merely

```
(SETQ NEXT-NUMBER (READ))
(SETQ SUM (IPLUS SUM NEXT-NUMBER))
```

4. The NEXT statement specifies the local variables that are to be updated for the next loop iteration. The ⟨next-case-expressions⟩ take the form of ⟨variable-expression⟩ pairs where the variable is set to the value of the expression. Typically, the expression involves some previous value of the variable in its computation. An example of a NEXT statement might be

```
(NEXT COUNT (ADD1 COUNT))
```

which is translated to

```
(SETQ COUNT (ADD1 COUNT))
```

5. The UNTIL statement terminates the execution of the loop when the ⟨until-condition⟩ evaluates to true (non-NIL). Stated differently, the

loop body is executed as long as the ⟨until-condition⟩ is false (NIL). The ⟨until-condition⟩ is just an S-expression that is evaluated. An example of an UNTIL statement might be

```
(UNTIL (IGREATERP SUM 1000))
```

which is translated to

```
(AND
     (IGREATERP SUM 1000)
     (GO EXIT$))
```

6. The RESULT statement specifies the value that the loop has when its termination conditions are satisfied. A ⟨result-expression⟩ is an S-expression that is evaluated and becomes the loop value. For example,

```
(RESULT (QUOTIENT SUM COUNT))
```

which translates to

```
(RETURN
     (QUOTIENT SUM COUNT))
```

Let us put the pieces together, with appropriate comments to see the framework for a general model of a PROG statement using the concepts discussed by Charniak et al. Our final PROG statement would appear as

```
(PROG      (SUM COUNT)
           (* from INITIAL statement *)
           (SETQ SUM 0)
           (SETQ COUNT 0)
LOOP$
           (* from WHILE statement *)
           (OR
                (NEQ COUNT 10)
                (GO EXIT$))
           (* from DO statement *)
           (SETQ NEXT-NUMBER (READ))
           (SETQ SUM (IPLUS SUM NEXT-NUMBER))
           (* from NEXT statement *)
           (SETQ COUNT (ADD1 COUNT))
           (* from UNTIL statement *)
           (AND
                (IGREATERP SUM 1000)
                (GO EXIT$))
           (GO LOOP$)
```

```
EXIT$
            (* from RESULT statement *)
            (RETURN
                 (QUOTIENT SUM COUNT)))
```

Note that Charniak's macros would insert the loop markers LOOP$ and EXIT$ automatically, but you must explicitly encode them in your PROG construct. We see that this PROG construct merely reads ten numbers (or until their sum is greater than 1000), and computes the average.

3.7.6 Other LISP forms

PROG provides us with two key features for writing programs:

1. The ability to perform iteration.
2. The ability to define and initialize local variables.

Other LISP dialects, such as MACLISP and FranzLisp, provide PROG expressions. But they also provide alternative forms that accomplish the same functions but with (they claim) simpler forms.

LET is an expression that allows you to declare and bind local variables. Its structure appears as

```
(LET ⟨local-variable-list⟩
     ⟨forms-list⟩)
```

where ⟨local-variable-list⟩ is a list of expressions of the form (⟨variable⟩ ⟨value⟩). ⟨forms-list⟩ is just a list of S-expression to be executed. The variables defined within a LET have existence only for the duration of the LET. You may think of a LET as a PROGN with local variable capability.

DO is a special form that provides the local variable binding capability of the LET form with the iteration facility of PROG. DO has the following structure:

```
(DO (⟨local-variable-list⟩)
    (⟨condition⟩ ⟨action-list⟩)
    (⟨expression[1]⟩
          ...
     ⟨expression[N]⟩))
```

The ⟨local-variable-list⟩ has a form similar to that used in LET expressions:

```
(⟨variable⟩ ⟨value⟩ ⟨update-expression⟩)
```

The ⟨update-expression⟩ is evaluated on each cycle of the DO loop to update the value of the variable. Variables with no ⟨update-expression⟩ are assumed to retain the initial value throughout the execution of the DO loop.

The (⟨condition⟩ ⟨action-list⟩) expression determines how the DO expression terminates. On each pass through the DO loop, the ⟨condition⟩ is executed. If it evaluates true, then the expressions on the ⟨action-list⟩ are executed and the DO expression is exited. The value of the DO expression is the value of the last expression in the ⟨action-list⟩.

The DO body is composed of expressions to be evaluated. Unlike PROG, however, when the DO reaches the last expression of the body, it begins a new cycle of the loop rather than falling off the end. The DO body may contain GO expressions and RETURN expressions to control the sequence of execution and when the loop is terminated, respectively. The DO body may be empty if all computation can be done by the ⟨update-expression⟩s in the ⟨local-variable-list⟩.

Charniak [char80], Winston [wins81], and Touretzky [tour84] discuss both the LET and DO forms in more detail.

Do not confuse the DO discussed in this section with the DO operator provided by CLISP.

3.8 VALUE ASSIGNMENT: SET AND SETQ

To assign values to variables, we use the **SETQ** function. SETQ takes two arguments: the variable to be set and an S-expression that may or may not be evaluated to provide a value for the variable. The format of SETQ (and SET as well) is

```
Function:       SETQ
                SET
                SETQQ

# Arguments:    2

Arguments:      1) a literal atom, ATM
                2) an S-expression, EXPRESSION

Value:          The new value of the variable.
```

ATM must be a literal atom. Attempting to use anything other than a literal atom causes the error message "ARG NOT LITATOM" to be displayed. If ATM is NIL or T, the error message "ATTEMPT TO SET NIL OR T" will be displayed.

SETQ and **SET** differ only in the evaluation of their first argument. SET evaluates its first argument to produce the name of the literal atom to be assigned a value. SETQ does not. That is, SETQ assumes that the first argument is the name of the variable to which a value is to be assigned.

Consider the following examples:

```
←(SETQ computer-manufacturers (LIST 'univac 'ibm 'ncr))
(univac ibm ncr)
```

```
←(SETQ an-industry 'computer-manufacturers)
computer-manufacturers

←(SET an-industry
       (APPEND computer-manufacturers
              (LIST 'honeywell)))
(univac ibm ncr honeywell)

←an-industry
computer-manufacturers

←computer-manufacturers
(univac ibm ncr honeywell)
```

Notice that SET evaluated the value of its first argument, which is COMPUTER-MANUFACTURERS, and set it to the value of its second argument. Thus, when we display COMPUTER-MANUFACTURERS later, it has the new value that is shown.

An alternative form of SETQ, **SETQQ**, assumes that both of its arguments are "quoted." That is, neither argument is evaluated. Using the example above, we would have

```
←(SETQQ computer-manufacturers (univac ibm ncr))
(univac ibm ncr)
```

3.9 SETTING AN ATOM'S VALUE CELL

INTERLISP has been implemented in several different versions. One major distinction concerns whether variables are deep or shallow bound on the stack. This notion will be discussed in more detail in Chapter 30. However, I will discuss several functions that set an atom's value cell in this chapter because they are so widely used, most notably by the File Package (see Chapter 16).

3.9.1 Binding Atoms from a File

RPAQ and **RPAQQ** are NLAMBDA functions that set an atom's value cell. They operate exactly like SETQ and SETQQ. They have the format

```
Function:       RPAQ
                RPAQQ
                RPAQ?

# Arguments:    2

Arguments:      1) a literal atom, ATM
                2) an S-expression, EXPRESSION

Value:          The value of the S-expression.
```

The expression is evaluated for RPAQ and is not evaluated for RPAQQ.

```
←(RPAQ computer-manufacturers '(univac ibm ncr))
(univac ibm ncr)
```

Both RPAQ and RPAQQ generate an error message "ARG NOT LITATOM" if ATM is not a literal atom.

RPAQ? sets the top level value of ATM if and only if it does not have a current top-level binding, e.g., the contents of the value cell is the atom NOBIND. It returns the value of EXPRESSION if the top level value is set, otherwise NIL.

RPAQ, RPAQQ, and RPAQ? are intended to be used from within the File Package.

These functions are often "hardwired" for efficiency because File Package operations are used so frequently by experienced INTERLISP programmers.

3.9.2 Getting and Setting the Top Level Value

As we mentioned in Section 2.1, atoms have value cells. When atoms are created by INTERLISP, they are allocated storage locations in memory. How a value is bound to an atom depends on the implementation:

1. Deep binding systems save the variable's new value on the stack. When a variable is referenced, its value is found by searching the stack for the most recent binding. If there is no binding on the stack, INTERLISP retrieves the value stored in the value cell of the atom.
2. Shallow binding systems save the variable name and old value on the stack, and place the new value in the atom's value cell. When a variable is accessed, the current value is always found in the value cell.

INTERLISP provides two pairs of functions for accessing the value of a variable. They take the forms

```
Function:        GETTOPVAL
                 GETATOMVAL

# Arguments:     1

Arguments:       1) an atom, ATM

Value:           The top level binding of ATM.

Function         SETTOPVAL
                 SETATOMVAL

# Arguments:     2

Arguments:       1) an atom, ATM
                 2) an S-expression, EXPRESSION

Value:           The value of the S-epxression.
```

GETTOPVAL returns the top level value of ATM, even if it is NOBIND, regardless of any other local bindings that may appear in the stack.

GETATOMVAL always returns the value cell contents of ATM. In shallow bound systems, it is equivalent to executing (EVAL atm). In deep bound systems, it defaults to GETTOPVAL.

SETTOPVAL sets the top level value of ATM regardless of other local bindings that may appear on the stack.

SETATOMVAL sets the value cell of ATM to the value of EXPRESSION. In a shallow bound system, it is equivalent to executing SET. In a deep bound system, it defaults to SETTOPVAL.

4

Fundamental Predicates

INTERLISP provides a large number of functions called **predicates**. A predicate is a function that tests some condition or attribute of its arguments. For example, a predicate may test whether its argument is an example of a given data type, whether it has a value satisfying certain criteria, or even whether it has a specific structure. The result of applying a predicate to its arguments is one of the atoms T or NIL (representing "true" or "false," respectively) or some non-NIL value representing true. Some predicates are known as *fundamental* predicates because they test essential characteristics of LISP objects. This chapter will discuss several of the fundamental predicates common to most LISP implementations. More predicates will be described in later chapters when we discuss the specific data types or subsystems with which they are associated.

By convention, the name of a predicate should always be terminated by the character "P" to indicate that it is a predicate. For historical reasons, some of the fundamental predicates do not follow this rule. As an INTERLISP programmer, it is good practice for you to terminate each application predicate function that you write with "P" to indicate that it is a predicate.

4.1 ATOM TESTING: ATOM AND LITATOM

The simplest datatype available in INTERLISP is the atom. Atoms are indivisible data structures. INTERLISP provides two predicates to test for the existence of atoms: ATOM and LITATOM.

```
Function:       ATOM
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          T if the argument is an atom; NIL
                otherwise.
```

ATOM determines whether or not its argument is an atom. If it is, ATOM returns T; otherwise, it returns NIL. ATOM is the most general predicate for testing atoms whether they be numbers or literal atoms. ATOM returns NIL if its argument is an instance of one of the other fundamental datatypes such as strings, arrays, etc. In some dialects of LISP, ATOM is defined to be equivalent to NLISTP (see Section 4.5), e.g., an atom is assumed to be anything that is not a list.

Another fundamental predicate is LITATOM. LITATOM tests whether or not its argument is a literal atom but not a number.

Function:	LITATOM
# Arguments:	1
Arguments:	1) an S-expression, EXPRESSION
Value:	T, if its argument is a literal atom; NIL otherwise.

Suppose that we let the value of the atom GIRL-FRIENDS be the following:

```
←(SETQ girl-friends
        '(jane nancy marcia susan ellen cheryl))
(jane nancy marcia susan ellen cheryl)
```

We can apply ATOM to this atom in the following ways:

Test if GIRL-FRIENDS is an atom

```
←(ATOM 'girl-friends)
T
```

Test the value of GIRL-FRIENDS

```
←(ATOM girl-friends)
NIL
```

Note that INTERLISP distinguishes between the name of an atom and its value. In this case, GIRL-FRIENDS is an atom when we test its name, but when we test its value, we see that its value is not an atom but a list.

We can apply ATOM to other arguments as follows:
To a number

```
←(ATOM 1378)
T
```

although this is a redundant test because we know that numbers are literal atoms.

To a string

```
←(ATOM "Sandra is beautiful")
NIL
```

because strings are not atoms, but a separate datatype.

Let us also apply LITATOM to several examples:

```
←(LITATOM 'girl-friends)
T
←(LITATOM 1378)
NIL
```

because numbers, while atoms, are not literal atoms.

```
←(LITATOM "Sandra is beautiful")
NIL
```

4.1.1 An Alternative Atomic Predicate

In other LISP dialects, the LITATOM predicate is replaced by the **SYMBOLP** predicate. SYMBOLP accomplishes the same function as LITATOM. Touretzky [tour84] notes that you may define SYMBOLP in terms of ATOM and NUMBERP (see below) as follow:

```
(DEFINEQ
     (symbolp (x)
          (AND
               (ATOM x)
               (NOT (NUMBERP x)))
     ))
```

where SYMBOLP will take the form

```
Function:       SYMBOLP
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          T, if its argument is a literal atom; NIL
                otherwise.
```

Let us apply SYMBOLP to a few examples to see how it works:

```
←(SYMBOLP 'girl-friends)
T
←(SYMBOLP 1378)
NIL
```

4.2 NUMERIC PREDICATES

INTERLISP provides several fundamental predicates for testing whether or not the value of an atom is a number. These include NUMBERP, ZEROP, FIXP, FLOATP, and SMALLP.

4.2.1 Testing for Numbers

NUMBERP returns its argument (meaning T) if the value of its argument is a number of any type; otherwise, it returns NIL. It takes the form

```
Function:       NUMBERP
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          T, if the value of its argument is a
                number; NIL otherwise
```

Consider the following cases in which NUMBERP is used:

```
←(SETQ a.number 1.7832)
1.7832
←(NUMBERP a.number)
1.7832 (which means T)
←(SETQ another-number 100)
100
←(NUMBERP another-number)
100   (which means T)
```

Note that NUMBERP works on both integers and floating point numbers with equivalent results.

```
←(NUMBERP ''1776")
NIL
←(NUMBERP 'FIVE)
NIL
```

NUMBERP works only with numeric representations of numbers. It does not know how to deal with string representations. We can circumvent this problem by using

```
←(NUMBERP (MKATOM "1776"))
1776
```

4.2.2 Testing for Zero

One of the most common tests that we make when performing arithmetic calculations is to determine if the value of some variable or expression is zero. INTERLISP provides a fundamental predicate for testing if the value of an atom or S-expression is zero: **ZEROP**. ZEROP returns T if the value of its argument is zero; otherwise, it returns NIL. It takes the form

```
Function:       ZEROP
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          T,   if the value of its argument is zero;
                NIL otherwise
```

Consider the following examples:

```
←(SETQ a-number 0)
0
←(ZEROP a-number)
T
```

Alternatively, we can test the value of an expression. For example, we can say

```
←(ZEROP (IPLUS 6 (IMINUS 6)))
T
```

where the value of the S-expression (IPLUS 6 (IMINUS 6)) is identically zero.

ZEROP returns NIL if its argument is not a number. For example,

```
←(ZEROP 'pi)
NIL
```

We should note that ZEROP is just a convenient function that could be defined in terms of other primitive functions. It is provided in most LISP systems

as a hardwired function for greater efficiency because it is so frequently used. A definition of ZEROP in terms of EQ would appear as

```
(DEFINEQ
    (ZEROP (number)
          (COND
               ((NUMBERP number)
                    (EQ number 0))
               (T NIL))
    ))
```

Most arithmetic functions generate an error if their argument is nonnumeric. Therefore, we test the argument to see if it is a number. If so, we compare it with 0 and return the result. Otherwise, we just return NIL.

Note: ZEROP should not be used for testing if floating point numbers are equal to zero because it uses EQ which works only on integers. You should use EQP instead. For example,

```
←(SETQ x 0)
0
←(EQ x 0.0)
NIL
←(EQP x 0.0)
T
←(EQUAL x 0.0)
T
```

4.2.3 A Generalized Zero Predicate

If you do not know whether your data will be integer or floating point numbers, you may want to define a generalized predicate for testing equality with zero. Let us call it **EQZERO**. It takes the following form

```
Function:       EQZERO
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          T, if its argument is zero either as a
                FIXP or a FLOATP; otherwise, NIL.
```

We might define EQZERO as follows

```
(DEFINEQ
   (eqzero (x)
```

```
(COND
    ((NUMBERP x)
       (COND
           ((FIXP x)
              (*
                   It is an integer!
              )
              (ZEROP x))
           ((FLOATP x)
              (*
                   It is floating point!
              )
              (EQP x 0.0))))
    (T
        (*
              It is not a number, so return NIL
        )
    NIL))
))
```

Consider the following examples:

```
←(SETQ anumber (IPLUS 6 (IMINUS 6))
0
←(EQZERO anumber)
T
←(SETQ anumber (FPLUS 6.0 (FMINUS 6.0))
0.0
←(EQZERO anumber)
T
←(EQZERO 'pi)
NIL
```

4.2.4 Testing the Type of Number

Numbers may be either integers or floating point numbers in INTERLISP. Two predicates allow you to test whether a number is an integer or a floating point number: **FIXP** or **FLOATP**. They take the following form

```
Function:       FIXP
                FLOATP

# Arguments:    1
```

```
Argument:       1) an S-expression, EXPRESSION
Value:          The value of EXPRESSION if it is a number
                of the specified type; otherwise, NIL.
```

Consider the following examples:

```
←(SETQ anumber 100.765)
100.765
←(FIXP anumber)
NIL
←(FLOATP anumber)
100.765
```

Both FIXP and FLOATP return NIL if their argument is not a number.

INTERLISP provides a predicate for testing whether or not a number is a small integer. Small integers arose from early implementations of LISP where space was extremely limited. The value of a small integer could be represented in the value cell itself rather than being pointed to by the contents of the value cell. **SMALLP** is the predicate that tests if a number is a small integer. It takes the form

```
Function:       SMALLP
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          The value of expression if it is a small
                integer; otherwise, NIL.
```

The range of small integers is implementation dependent and is discussed further in Chapter 13. Consider the following examples (on INTERLISP-10):

```
←(SMALLP 25)
25
←(SMALLP 33762)
NIL
```

because the range of small integers on INTERLISP-10 is -1535 to 1535. Other implementations will have different ranges.

4.3 STRING TESTING: STRINGP

INTERLISP provides a fundamental predicate for testing whether or not the value of its argument is a string. This predicate is called **STRINGP**. STRINGP takes the form

```
Function:       STRINGP

# Arguments:    1

Argument:       1) an expression, EXPRESSION

Value:          T, if the expression has the datatype
                string; NIL otherwise.
```

STRINGP returns the string if the value of its argument has a datatype of STRING; NIL otherwise.

Consider the following examples:

```
←(SETQ astring
       "The quick brown fox jumped over the lazy dog")
"The quick brown fox jumped over the lazy dog"

←(STRINGP astring)
"The quick brown fox jumped over the lazy dog"

←(STRINGP 1.7875)
NIL

←(STRINGP 'presidents)
NIL

←(STRINGP (MKSTRING 1776))
"1776"
```

4.4 ARRAY TESTING: ARRAYP

INTERLISP provides a fundamental predicate, **ARRAYP**, for testing whether or not its argument has a datatype of ARRAY. ARRAYP takes one argument—a potential array object. It determines whether or not the argument is an array object. If it is, it returns the value of the array object. If it is not, it returns NIL. ARRAYP takes the form

```
Function:       ARRAYP
                HARRAYP

# Arguments:    1

Argument:       1) An address of an array, ARRAYX

Value:          The value of the address if it is a
                pointer to an array object;
                otherwise, NIL.
```

HARRAYP returns the address of a hash array if its argument has the datatype hash array.

Note: arrays may only be created by the function ARRAY (see Section 11.1). Arrays have a special format used by INTERLISP to manage their contents that requires allocation from a special storage pool. Arrays are discussed in more detail in Chapter 11.

4.5 LIST TESTING: LISTP and TAILP

INTERLISP provides a fundamental predicate to test whether or not its argument is a list. This predicate is called **LISTP**. It takes the form

```
Function:       LISTP
                NLISTP

# Arguments:    1

Argument:       1) an S-expression, EXPRESSION

Value:          T, if value of argument is a list; NIL
                otherwise
```

LISTP returns T if its argument is a list; otherwise, it returns NIL. Consider the following examples:

```
←(SETQ presidents
       (CONS 'kennedy
             (CONS 'johnson 'nixon)))
(kennedy johnson nixon)

←(LISTP presidents)
T
```

where the list is built by multiple CONSes.

```
←(SETQ presidents "kennedy johnson nixon")
"kennedy johnson nixon"

←(LISTP presidents)
NIL

←(SETQ numbers
       (ARRAY 5 3 0))
[ARRAYP]#1,2120

←(SETA numbers 1 100)
100

←(SETA numbers 2 200)
200
```

```
←(SETA numbers 3 300)
300
←(LISTP numbers)
NIL
```

Neither arrays nor strings are represented as lists in INTERLISP. Thus, LISTP returns a value of NIL because they are not list-structures. For example,

```
←(LISTP NIL)
NIL
```

because NIL is considered to be a literal atom, rather than a list. Thus, while (LITATOM NIL) = (ATOM NIL) = T, LISTP applied to NIL returns NIL. You should exercise caution if the value of an object may be the empty list.

Note that other implementations of LISP may consider NIL ("the empty list") to represent a list whence LISTP will return T. This may cause difficulty when you attempty to transport source code from one dialect of LISP to another.

An alternative form, **NLISTP**, returns the logical negation of LISTP, i.e., it asks if its argument is not a list. We might define NLISTP as

```
(DEFINEQ
     (nlistp (x)
          (NULL (LISTP x))
     ))
```

Consider the following example:

```
←(NLISTP 'presidents)
T
←(NLISTP presidents)
NIL
```

where PRESIDENTS has the value (kennedy johnson nixon).

4.5.1 Testing for the Tail of a List

We have already seen how to take lists apart using the CAR and CDR functions. We often want to test whether or not the remainder of a list (that is, its *tail*) has a certain value. With CAR and CDR we can test the permissible combinations of these functions. However, most lists tend to be longer than four elements. Dismembering such lists can be difficult if we have to write specific functions to take a given number of CDRs from a list. INTERLISP provides us with a convenient function to test the tail of a list for equivalence to a specific value, **TAILP**.

The generic format of this function is

```
Function:       TAILP
# Arguments:    2
Arguments:      1) a list structure, X
                2) any list structure, LST
Value:          X, if the value of X is EQ to some number
                of CDRs (>=0) of LST; NIL otherwise.
```

TAILP returns the value of its first argument if that argument is a tail of its second argument; otherwise, it returns NIL. A tail is defined as the list resulting from taking some number of CDRs of the second argument. Consider the following examples:

```
←(SETQ girl-friends
       '(marcia mary janice angela elizabeth))
(marcia mary janice angela elizabeth)
←(TAILP '(angela elizabeth) girl-friends)
NIL
```

because TAILP uses EQ to compare the two lists. In this case, '(angela elizabeth) is a new list with a new storage allocation. On the other hand,

```
←(SETQ recent-girl-friends (CDDDR girl-friends))
(angela elizabeth)
←(TAILP recent-girl-friends girl-friends)
(angela elizabeth)                          which means T
```

because some number of CDRs of GIRL-FRIENDS returns a list that is EQ to the value of its first argument.

The value of the first argument X must match exactly the structure of some tail of the second argument LST. Using EQ, the value is T if the value of X is a substructure of LST. That is, for TAILP to succeed, X must be embedded in LST such that the last N elements of LST correspond exactly to the elements of X.

Thus, the following examples will not succeed:

```
←(SETQ acquaintance '(sandra))
(sandra)
←(TAILP acquaintance girl-friends)
NIL
```

because no number of CDRs produces a list that has the same value as a tail of GIRL-FRIENDS.

```
←(SETQ a-good-friend '(elizabeth))
(elizabeth)

←(SETQ one-good-friend (CDDDDR girl-friends))
(elizabeth)

←(TAILP a-good-friend girl-friends)
NIL
```

because A-GOOD-FRIEND does not point to the same structure in memory as the element ELIZABETH in the list GIRL-FRIENDS.

```
←(TAILP one-good-friend girl-friends)
(elizabeth)
```

because ONE-GOOD-FRIEND points to a list structure that is embedded within the second argument.

For mathematical purposes, we say that X is a *proper tail* of LST if the number of CDRs necessary to find the tail is greater than zero. Unfortunately, no indication of the number of CDRs required to determine the tail is provided by INTERLISP.

We might define TAILP as follows

```
(DEFINEQ
     (tailp (x lst)
          (AND x
               (PROG NIL
               loop
                    (COND
                        ((NLISTP lst)
                             (RETURN NIL))
                        ((EQ x lst)
                             (RETURN x)))
                    (SETQ lst (CDR lst))
                    (GO loop)))
     ))
```

4.5.2 Counting the CDRs To Produce a Tail

Suppose you wanted to know how many CDRs it will take to reach the tail of LST. We could modify the definition of TAILP to return this information instead of the value of the tail. Let us call this function **TAILP?** and let it take the form

```
Function:       TAILP?

# Arguments:    2

Arguments:      1) an S-expression, X
                2) a list structure, LST

Value:          The number of CDRs required to reach the
                tail of LST if X satisfies TAILP on LST;
                otherwise, NIL.
```

We might define TAILP? as follows

```
(DEFINEQ
     (tailp? (x lst)
          (AND x
               (PROG (ntails)
                     (SETQ ntails 0)
               loop
                     (COND
                         ((NLISTP lst)
                              (RETURN NIL))
                         ((EQ x lst)
                              (RETURN ntails)))
                     (SETQ lst (CDR lst))
                     (SETQ ntails (ADD1 ntails))
                     (GO loop)))
     ))
```

Note that TAILP? returns zero if X and LST are exactly equal; that is, no CDRs are required to produce the tail. If no number of CDRs would produce a tail because X is not a tail of LST, then TAILP? returns NIL. Otherwise, it counts the number of CDRs and returns that value.

4.6 TESTING FOR EQUALITY

In any programming language, we want to be able to test the equality of two objects. INTERLISP provides predicates for testing equality based on the data-type of the objects as well as the equality of certain attributes of the objects.

4.6.1 EQ versus EQUAL

The two basic predicates for testing equality are **EQ** and **EQUAL**. They have the following format

```
Function:       EQ
                EQUAL

# Arguments:    2

Arguments:      1) a LISP object, X
                2) a LISP object, Y

Value:          T, if the objects are equal; NIL,
                otherwise.
```

EQ and EQUAL differ in the way they compare their arguments. Consider two literal atoms that have the same name. Both EQ and EQUAL will return T. For example,

```
←(EQ 'john 'john)
T
←(EQUAL 'john 'john)
T
```

both of which return T because each literal atom points to a unique location in storage. Thus, there is only one instance of JOHN in the entire INTERLISP memory.

EQ and EQUAL produce the same result when their arguments have the same values if those values are literal atoms or numbers. For example,

```
←(SETQ name-1 'john)
john
←(SETQ name-2 'john)
john
←(EQ name-1 name-2)
T
←(EQUAL name-1 name-2)
T
```

because the values of NAME-1 and NAME-2 are the same literal atom.

EQ operates by comparing the pointers of its arguments. If they point to the same structure, EQ returns T. Otherwise, it returns NIL. However, a problem arises when we apply EQ to more complex data structures such as lists and strings. Consider the following example

```
←(SETQ languages-i-know
        (LIST 'pascal 'fortran 'snobol 'cobol))
(pascal fortran snobol cobol)
```

```
←(SETQ languages-used
        (LIST 'pascal 'fortran 'snobol 'cobol))
(pascal fortran snobol cobol)

←(EQ    languages-i-know languages-used)
NIL

←(EQUAL languages-i-know languages-used)
T
```

In this case, EQ returns NIL while EQUAL returns T. Why? We must remember that each time LIST is applied to its arguments, it consumes additional storage to construct and return a new list. Thus, although the two lists LANGUAGES-I-KNOW and LANGUAGES-USED have the same elements, they are stored in different locations in memory. Because EQ uses pointers to data structures for comparison, the two lists are not equal because they have different addresses in memory.

To test the equality of two data structures, we must use EQUAL, which compares the elements of the two structures. Thus, EQUAL returns T because it compares the successive elements of the two lists LANGUAGES-I-KNOW and LANGUAGES-USED, and finds them equivalent.

EQUAL compares the top-level values of its arguments. Thus, two structures will be EQUAL if

1. EQ — That is, pointers to the same structure.
2. EQP — That is, numbers with equal value.
3. STREQUAL — That is, strings containing the same sequence of characters.
4. Lists whose CARs are EQUAL and whose CDRs are EQUAL, applied recursively.

EQ and EQUAL also work correctly when given pointers to the same array object. However, EQUAL returns NIL if it is asked to compare two different arrays because it does not perform an individual comparison of the array elements.

Why should you use EQ over EQUAL? Basically, it is a matter of efficiency. EQ is a primitive operation that may be encoded as a single instruction whereas EQUAL must always be defined as a subroutine because it has more work to do. On the other hand, if you are not too worried about the efficiency of your program, but are worried about it performing the proper checks every time, then I would encourage you to use EQUAL wherever possible.

Programming Convention: It is a good idea, if you are a novice, to always use EQUAL to ensure that the proper evaluation of the two arguments is performed by INTERLISP. Because data structures that are EQUAL are not always EQ, many beginning programmers expend substantial effort in attempting to

determine why a program does not work correctly even though the data appear to be exactly the same.

Note that some dialects of LISP allocate new storage every time a number is created even though that number may already be represented in memory. Thus, two numbers may never be guaranteed to be EQ, even if they are EQUAL.

4.6.2 Atomic Equality

When we have complex data structures, we may want to know if they are exactly equal throughout the entire structure. To determine total equality, we must descend to the atomic level throughout the data structure. EQUAL does not operate in this manner since it compares only top-level values. To thoroughly test two data structures, INTERLISP provides the predicate EQUALALL.

EQUALALL always descends to the atomic level of each of its arguments to determine equality. Thus, it should be used when comparing arrays, user data types, or complex structures having multiple levels of sublists beneath the top level. It takes the form

```
Function:       EQUALALL

# Arguments:    2

Arguments:      1) a data structure, X
                2) a data structure, Y

Value:          T, if each element of X is EQUAL to the
                corresponding element of Y; otherwise,
                NIL.
```

EQUALALL may be used to determine the equality of two arrays by inspecting their contents. Let us define two arrays as follows

```
←(SETQ A1 (ARRAY 5 5))
[ARRAYP]#542224
←(SETQ A2 (ARRAY 5 5))
[ARRAYP]#542233
```

Clearly, EQ and EQUAL will not work on the values of A1 and A2, respectively, because they are different addresses for arrays. Let us initialize the arrays as follows (using a CLISP expression):

```
←(FOR I FROM 1 TO 5
      DO
         (SETA A1 I (ITIMES I 100))
         (SETA A2 I (ITIMES I 100)))
NIL
```

```
←(FOR I FROM 1 TO 5
      DO
         (PRIN1 (ELT A1 I))
         (TAB 20)
         (PRINT (ELT A2 I)))
100                  100
200                  200
300                  300
400                  400
500                  500
NIL
```

Now, let us compare A1 and A2 for equality using EQUALALL:

```
←(EQUALALL A1 A2)
T
```

because EQUALALL descends into the array to compare individual elements.

Now, let us compare two lists which have equivalent elements but are not EQ or EQUAL. First, let us define the lists

```
←(SETQ LST1
       (LIST 'red
             'yellow
             (LIST 'green 'blue)
             (LIST 'black)))
(red yellow (green blue) (black))
←(SETQ LST2
       (LIST 'red
             'yellow
             (LIST 'green 'blue)
             (LIST 'black)))
(red yellow (green blue) (black))
```

These two lists occupy different locations in memory because LIST consumes new CONS cells each time it is called. Thus, we may compare the two lists for equality

```
←(EQ lst1 lst2)
NIL
←(EQUALALL lst1 lst2)
T
```

4.6.3 Numeric Equality

EQ returns T if two numbers have the same structure in memory. For example,

```
←(SETQ ten 10)
10
←(EQ   10 ten)
T
```

However, EQ cannot compare an integer with a floating point number even though they may have the same value. For example,

```
←(SETQ ten 10)
10
←(SETQften (FLOAT ten))
10.0
←(EQ ten ften)
NIL
```

INTERLISP provides **EQP** to test numerical equality between two numbers. However, EQP does not do conversion to a canonical representation. Thus, EQP will succeed when

```
←(EQP ten ften)
T
```

but will fail when

```
←(EQP ten 10.3)
NIL
```

EQP takes the format

```
Function:       EQP
# Arguments:    2
Arguments:      1) a number, X
                2) a number, Y
Value:          T, if the two numbers are equal;
                otherwise, NIL.
```

EQP may be used to compare X and Y as objects. It returns T if X and Y are EQ; NIL, otherwise. X and Y may be array pointers or stack pointers.

4.6.4 Testing Equality of Length

In many applications, you will want to know if the length of a list is at least equal to a given number. INTERLISP provides the predicate **EQLENGTH** to test the size of a list. The format of EQLENGTH is

```
Function:       EQLENGTH

# Arguments:    2

Arguments:      1) an S-expression, EXPRESSION
                2) a minimum length, LEN

Value:          T, if the S-expression has the minimum
                length; NIL, otherwise
```

A simple definition of EQLENGTH might appear as

```
(DEFINEQ
     (eqlength (expression len)
          (IGEQ (LENGTH expression) len)
     ))
```

Whatever the length of EXPRESSION, LENGTH must traverse the entire structure to determine its length before the comparison of values may take place. If EXPRESSION is very long, substantial time may be consumed in determining its length. Moreover, if EXPRESSION is a circular list, LENGTH never terminates until the operating system steps in.

A more efficient version of EQLENGTH that is safe to use with circular lists can be defined as

```
(DEFINEQ
     (eqlength (expression alength)
          (PROG (alst the-length)
                (SETQ alst expression)
                (SETQ the-length 0)
           LOOP
                (COND
                    ((CAR alst)
                         (*
                              If there is a CAR cell,
                              increment the counter for
                              the length of the list.
                         )
                         (SETQ the-length (ADD1 the-
length))
                         (SETQ alst (CDR alst))
                    (T
```

```
                    (RETURN NIL)))
            (COND
                ((EQUAL the-length alength)
                    (*
                        Exit with T if and only if
                        the list has a length equal
                        to ALENGTH.
                    )
                    (RETURN T)))
            (GO LOOP))
    ))
```

In this definition, EQLENGTH will terminate when ALST is determined to have the minimum length specified or ALST is determined to have a length less than that specified. It works for circular lists because the number of comparisons is bounded by ALENGTH.

4.6.5 Testing Complex or Circular Structures

Some applications may require the use of circular structures (although this is not generally recommended). Other applications require complex structures where we want to know only if they are equal to some depth of recursion. INTERLISP provides the predicate **EQUALN** to test if two structures are equal to a given depth. Its format is

```
Function:       EQUALN

# Arguments:    3

Arguments:      1) an S-expression, EXP1
                2) an S-expression, EXP2
                3) a depth, DEPTH

Value:          T, if EXP1 equals EXP2 to the given depth
                and no further recursion is possible;
                ?, if EXP1 equals EXP2 and further
                recursion is possible; NIL, otherwise.
```

EQUALN uses DEPTH to determine how deep to search in the complex structure. For example,

```
←(EQUALN '(((a)) b) '(((z)) b) 2)
?
```

because at level 2 of the CAR recursion, it still has to compare (A) to (Z). At level 2 they are equal, but further recursion remains, so equality is undetermined.

```
←(EQUALN '(((a)) b) '(((z)) b) 3)
NIL
```

With the depth were set to 3, EQUALN would return NIL because it would compare A to Z and find them not equal.

```
←(EQUALN '(((a)) b) '(((a)) b) 3)
T
```

Note that DEPTH may be set to 0. Consider the following simple test cases:

```
←(EQUALN '(A) '(A) 0)
?
←(EQUALN '(A) '(A) 1)
T
```

4.6.6 Testing for Non-Equality

Just as we may test for equality, we may also test for non-equality. INTERLISP provides the predicate **NEQ** to determine if two data structures are not equal to each other. This test is very simple because INTERLISP uses EQ to compare the pointers to the two structures. NEQ returns T if the two structures are not equal; otherwise, it returns NIL. It takes the form

```
Function:       NEQ
                NOTEQUAL

# Arguments:    2

Arguments:      1) an S-expression, EXPRESSION1
                2) an S-expression, EXPRESSION2

Value:          T, if EXPRESSION1 is not equal to
                EXPRESSION2; NIL, otherwise.
```

A simple definition of NEQ might appear as

```
(DEFINEQ
     (neq (expression1 expression2)
          (NOT (EQ expression1 expression2))
     ))
```

Note that because NEQ uses EQ to compare the two data structures, the two data structures may be EQUAL without being EQ. Thus, you may also want to define a function NOTEQUAL which ensures that they are not equal at the top level.

A simple definition of NOTEQUAL might appear as

```
(DEFINEQ
     (notequal (x y)
          (NOT (EQUAL x y))
     ))
```

Note that you may also want to define a function NOTEQUALALL which determines that two structures are not equal at the atomic level.

4.6.7 Testing for Null

INTERLISP provides the predicate **NULL** to test whether or not its argument has the value NIL. NULL returns T if the argument has the value NIL. Otherwise, it returns NIL. Its format is

```
Function:       NULL
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          T, if the value of EXPRESSION is NIL; NIL,
                otherwise.
```

Consider the following examples:

```
←(NULL (LIST))
T
```

because NIL is treated as a literal atom.

```
←(NULL (CONS))
NIL
```

4.7 TESTING VARIABLE BINDINGS

Some functions use "free" or "global" variables in their computations. To prevent errors, you may want to determine whether or not a variable has been bound to a value before proceeding with the computation. INTERLISP provides the predicate **BOUNDP** to test if a variable is bound to anything in the current context. It takes the form

```
Function:       BOUNDP
# Arguments:    1
```

```
Argument:       1) a variable, VAR
Value:          T, if the variable is bound in some
                environment.
```

BOUNDP returns T if its argument has a top level value. If the value of the variable is NOBIND or if the variable has not been created, BOUNDP will return NIL. Consider the following example

```
...
(PROG (var-1 var-2 var-3 ...)
      (SETQ var-1 ⟨some value⟩)
      (SETQ var-2 NIL)
      ...
      (⟨a-function⟩ var-1 var-2)
      ...
      (RETURN))
      ...
```

Note that in the PROG we have bound VAR-1 and VAR-2 to some values. However, we have not bound VAR-3. If we were to use VAR-3 in A-FUNCTION, an error would result (specifically, U.B.A.) because VAR-3 has not been given a value. The error results when INTERLISP attempts to (EVAL var-3) to determine its value.

We can avoid this error by using BOUNDP to determine if a variable is bound before it is used. Thus, in A-FUNCTION, we might encode a condition as follows

```
...
(COND
     ((NOT (BOUNDP var-3))
          (SETQ var-3 NIL)))
```

which sets VAR-3 to NIL if it does not have a value.

Consider the following example (assuming we have never set X to any value)

```
←(BOUNDP 'X)
NIL
←(SETTOPVAL 'X 'ABC)
ABC
←(BOUNDP 'X)
T
```

4.8 DETERMINING MEMBERSHIP IN A LIST

In many applications, a list represents a collection of related items. Often, we want to know whether or not another item is a member of that set of items. **MEMBER** determines if an item X is a member of a list. It takes the form

```
Function:       MEMBER
                MEMB
                EQMEMB

# Arguments:    2

Arguments:      1) an element, X
                2) a list of elements, LST

Value:          The tail of LST beginning with X if X is a
                member of LST; NIL, otherwise.
```

MEMBER uses EQUAL to compare X against the elements of LST. Consider the following example,

```
←(SETQ presidents '(reagan carter ford nixon johnson
...))
(reagan carter ford nixon johnson ...)
```

assuming that the list PRESIDENTS contains the last names of all the presidents of the United States.

```
←(MEMBER 'disney presidents)
NIL
←(MEMBER 'adams presidents)
(adams monroe madison jefferson adams washington)
```

Note that PRESIDENTS contains the name ADAMS twice, representing the sixth and second presidents respectively. However, because we have ordered the list beginning with the most recent president, the first occurrence of ADAMS will be detected in the list.

We might define MEMBER as follows

```
(DEFINEQ
     (member (x lst)
          (PROG NIL
          loop
                (COND
                    ((NLISTP lst)
```

```
                    (*
                        If LST is not a lst, then
                        return NIL
                    )
                    (RETURN NIL))
                ((COND
                    ((LITATOM x)
                        (*
                            If X is a literal
                            atom, use EQ for speed
                            in comparing.
                        )
                        (EQ x (CAR lst)))
                    (T
                        (*
                            Otherwise, compare X
                            with the top level
                            components of the
                            first element of LST.
                        )
                        (EQUAL x (CAR lst))))
                    (RETURN lst)))
            (SETQ lst (CDR lst))
            (GO loop))
    ))
```

An alternative form, **MEMB**, uses EQ instead of EQUAL to perform its comparisons. MEMB should only be used if you are certain that the elements of list may be uniquely compared. That is, the elements of the list should either be numbers, T or NIL, or literal atoms.

For MAKEFILE (see Section 17.3.1), you may specify a list of options that determines how the code that is written to the file will be processed. MAKEFILE checks the options that you specify to determine if they are valid, and sets certain flags that are used later in the function. An abstract of the code used to check flags is shown below

```
(COND
    ((MEMB 'NOCLISP OPTIONS)
        (RESETSAVE PRETTYTRANFLG T)))
(COND
    ((MEMB 'FAST OPTIONS)
        (RESETSAVE PRETTYTRANFLG NIL)))
(COND
    ((OR
        (MEMB ''CLISPIFY OPTIONS)
```

```
        (MEMB 'CLISP OPTIONS))
        (RESETSAVE CLISPIFYFLG T)))
```

Because MEMBER uses EQUAL, it will determine if two objects are equal even if they do not have the same memory location. MEMB, however, uses EQ which checks for equality of location. I suggest that you use MEMBER (even though it takes more time) when you are inspecting a list for an element.

Another form, **EQMEMB**, is T if either X is equal to LST or X is a member of LST. In the definition of MEMBER given above, if LST is not a list, then MEMBER returns NIL. EQMEMB allows us to specify either an atom or a list as its second argument. If the second argument is an atom, EQMEMB determines if it is identical to the first argument; otherwise, it invokes MEMBER with the two arguments. We might define EQMEMB as follows:

```
(DEFINEQ
    (eqmemb (x y)
        (COND
            ((nlistp y)
                (EQ x y))
            (T
                (MEMBER x y)))
    ))
```

Logical Connectives and Predicates

INTERLISP provides a number of functions for performing logical functions on collections of S-expressions. These logical functions include the Boolean AND, OR, and NOT, and several Boolean predicates including SOME and EVERY. You should note that the arguments to each of these functions are evaluated prior to application of the logical function to the set of results produced by those evaluations.

5.1 LOGICAL CONJUNCTION

A conjunction is an expression whose value is true if and only if each of its components evaluates to a true value.

The logical **AND** function accepts an indefinite number of S-expressions as its arguments; the number of arguments may be zero. It takes the form

```
Function:       AND

# Arguments:    0-N

Arguments:      1-N) one or more S-expressions,
                     EXPRESSION[1] ... EXPRESSION[n]

Value:          The value of the last argument, if all of
                its S-expressions evaluate to T or its
                equivalent; otherwise, NIL.
```

AND is an NLAMBDA, nospread function. Each of the arguments to AND is evaluated in turn until one such argument is determined to have a NIL value. At that time, the AND terminates, returning NIL. If all of its arguments have non-null values, AND returns the value of the last argument. For example,

```
←(AND '(I CAME) '(I SAW) '(I CONQUERED))
(I CONQUERED)
```

Consider the following examples:

No arguments

```
←(AND)
T
```

because no argument is detected as having a value of NIL.

First argument is NIL

```
←(AND NIL ⟨anything⟩)
NIL
```

always results in NIL because its first argument is NIL.

Some argument is NIL

```
←(SETQ presidents '(kennedy johnson nixon ford))
(kennedy johnson nixon ford)
←(AND
     (MEMBER 'nixon presidents)
     (MEMBER 'ford presidents)
     (MEMBER 'disney presidents))
NIL
```

When the arguments to AND are evaluated, we see that the first two S-expressions succeed because NIXON and FORD are members of the list PRESIDENTS. However, upon evaluating the third S-expression, a value of NIL is returned because DISNEY is not a member of PRESIDENTS. Thus, a value of NIL is returned as the value of the AND function. The order of the S-expressions makes no difference in this example (but will below), so we can rewrite the above example as

```
←(AND
     (MEMBER 'nixon presidents)
     (MEMBER 'disney presidents)
     (MEMBER 'ford presidents))
NIL
```

In this case, the last S-expression will not be evaluated because the second S-expression has a value of NIL which terminates the execution of the AND function.

No arguments are NIL

```
←(AND
     (MEMBER 'nixon presidents)
     (MEMBER 'ford presidents)
```

```
    (MEMBER 'kennedy presidents))
(kennedy johnson nixon ford)
```

In this example, all three S-expressions have non-null values. Therefore, the AND function returns as its value the value of the last S-expression [because MEMBER returns a non-null value if it succeeds (see Section 4.8)].

5.1.1 An Application of AND

The AND function may be used in many interesting ways. One interesting form has been to simulate the action of an UNTIL clause in a DO...UNTIL loop [char80].

The WHILE...DO... loop formalism is created using the PROG function (see Section 3.7). A skeletal form would appear as follows

```
(PROG     (⟨variables⟩)                          (* INITIAL *)
          (SETQ ⟨variable⟩ ⟨expression⟩)
                .
                .
                .
          (SETQ ⟨variable⟩ ⟨expression⟩)          (* INITIAL *)
LOOP
          (OR                                    (* WHILE *)
              ⟨expressions⟩
              (GO EXIT))
          ⟨expressions⟩                          (* DO *)
          (SETQ ⟨variable⟩ ⟨expression⟩)            (* NEXT *)
          (AND                                   (* UNTIL *)
              ⟨expressions⟩
              (GO EXIT))
          (GO LOOP)
EXIT
          (RETURN ⟨expression⟩))                  (* RESULT *)
```

A detailed analysis of this formalism was discussed during the examination of the PROG function (see Section 3.7.5).

When the AND function is reached in this formalism, we have already executed all of the ⟨expressions⟩ that correspond to the DO-body of an iterative loop. At that time, the ⟨expressions⟩ within the AND function are evaluated. If any of these ⟨expressions⟩ have a value of NIL, we jump to execute the loop once more. When all ⟨expressions⟩ have non-null values, we exit from the loop.

5.2 LOGICAL DISJUNCTION

A disjunction is an expression whose value is true if and only if at least one of its components evaluates to a true value.

The logical **OR** function operates in a fashion similar to the logical AND function. It also takes an indefinite number of arguments each of which is evaluated in turn. It takes the form

```
Function:       OR
# Arguments:    0-N
Arguments:      1-N) one or more S-expressions,
                EXPRESSION[1] ... EXPRESSION[n]
Value:          The value of the first non-NIL argument,
                if any of its arguments evaluates to T or
                its equivalent; otherwise, NIL.
```

OR is an NLAMBDA, nospread function. However, OR returns the value of the first S-expression whose value is non-null; otherwise, it returns NIL if all its arguments have a value of NIL. Consider the following examples:

No arguments

```
←(OR)
NIL
```

because no argument has a non-null value.

First Argument is non-null

```
←(OR T ⟨anything⟩)
T
```

because the first argument is always non-null, it returns T.

Some argument is non-null

```
←(OR
     (MEMBER 'nixon presidents)
     (MEMBER 'disney presidents))
     (nixon ford)
```

returns the list (NIXON FORD) because NIXON is a member of the list PRESIDENTS and MEMBER always returns the tail of the list consisting of the matching element.

No argument is non-null

```
←(OR
     (MEMBER 'disney presidents)
```

```
(MEMBER 'churchill presidents))
NIL
```

because none of the S-expressions has a non-null value.

5.2.1 An Application of the OR Function

Reviewing the sample structure given in Section 5.1.1, you will see that OR is used to select a terminating condition for the loop. In this case, OR implements the WHILE phrase of the loop by allowing the iteration to continue if any of its arguments is non-NIL.

We often combine AND and OR in expressions to test different conditions. In Section 4.6.3, we used EQP to test the equality of two numbers. Suppose that we do not care whether the numbers are equal, but do care about the equality of their signs. The function EQSIGNP allows us to test this concept. It is defined as

```
(DEFINEQ
    (eqsignp (x y)
        (OR
            (AND
                (ZEROP x)
                (ZEROP y))
            (AND
                (LESSP x 0)
                (LESSP y 0))
            (AND
                (GREATERP x 0)
                (GREATERP y 0)))
        ))
```

The first argument to OR tests a simple case (it executes more quickly than either of the following relational comparisons). It also handles the case where X is 0 and Y is 0.0. The latter two arguments to OR handle the cases where both arguments to EQSIGNP are less than or greater than 0.

5.3 LOGICAL NEGATION

Logical negation has the effect of returning the opposite truth value of its argument. INTERLISP provides two functions to perform logical negation: NOT and NEGATE.

5.3.1 Computing the Logical Negation

NOT returns the opposite value of its argument. That is, it turns T (or some non-NIL value) into NIL, and NIL into T. It takes the form

```
Function:       NOT
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          The logical negation of the value of its
                argument.
```

The value given to NOT may be any S-expression, including atoms and lists. For example,

```
←(NOT (LIST 'michigan 'illinois 'purdue))
NIL
```

Many of the predicates in INTERLISP do not have the corresponding opposite forms. Two which do are NLISTP, which is the opposite of LISTP, and NEQ, which is the opposite of EQ. As we saw in Section 4.6.6, we may create the opposite predicate by defining a function that prepends NOT to an existing predicate. In general, you will find these forms easier to read and understand than writing an S-expression of the form

```
(NOT (⟨predicate⟩ ⟨arguments⟩))
```

wherever you want to use the opposite predicate. There is a minimal cost to this approach, namely, defining a new function with the proper name and definition.

5.3.2 Creating Negated S-expressions

NOT merely computes the opposite truth value of the value of its argument. **NEGATE**, on the other hand, returns an S-expression that would compute the opposite value. It takes the form

```
Function:       NEGATE
# Arguments:    1
Arguments:      1) an S-expression, EXPRESSION
Value:          The S-expression evaluating to the logical
                negation of the value of its argument.
```

NEGATE returns an S-expression which will evaluate to the negative of a value. For example,

```
←(NEGATE 'x)
(NOT x)
```

```
←(SETQ x T)
T
←(NEGATE x)
NIL
```

NEGATE is written to inspect the structure of its argument and perform some simple Boolean manipulations to generate the resulting form. Consider the following examples:

```
←(NEGATE '(AND x y))
(OR (NOT x) (NOT y))
```

Suppose we assign the values T and NIL to X and Y respectively. Then we may see how NEGATE operates:

```
←(SETQ x T)
T
←(SETQ y NIL)
NIL
←(AND x y)
NIL
←(NEGATE '(AND x y))
(OR (NOT x) (NOT y))
←(OR (NOT x) (NOT y))
T
```

The corresponding form using OR is

```
←(NEGATE '(OR x y))
(AND (NOT x) (NOT y))
```

NEGATE recognizes the negative predicates that are defined within INTERLISP. For example,

```
←(NEGATE '(EQ x y))
(NEQ x y)
←(NEGATE '(OR X (NLISTP y)))
(AND (NOT x) (LISTP y))
```

If NEGATE cannot resolve the form into a series of predicates it knowns about, it prepends NOT to the value of EXPRESSION and returns that as its

value. Thus, if we had defined NOTEQUAL as a function (see Section 4.6.6), NEGATE does not know about it. Thus, we have the following case:

```
←(NEGATE '(EQUAL x y))
(NOT (EQUAL x y))
```

NEGATE works with any number of logical operators composed in an S-expression, but only at the top level of the expression. Consider the following complex expression:

```
←(NEGATE '(OR (NOT b) (AND a (NOT (OR b d)))))
(AND b (OR (NOT a) (OR b d)))
```

5.4 UNIVERSAL QUANTIFICATION

In predicate logic, a formula P(x) may have the value T no matter what assignment is given to the variable X. To assert that every value of X chosen from some domain satisfies the predicate P, we place a *universal quantification* symbol naming the variable in front of the predicate. INTERLISP provides a function to test whether or not all members of a list satsify a given predicate.

EVERY tests if the application of a given function to each element of a list results in a value of T; otherwise, it returns NIL. It takes the form

```
Function:       EVERY
                NOTEVERY

# Arguments:    3

Arguments:      1) any S-expression, EVERYX
                2) an evaluation function, EVERYFN1
                3) a selection function, EVERYFN2

Value:          T if (EVERYFN1 (CAR EVERYX)) is T for all
                elements of EVERYX selected by EVERYFN2;
                NIL otherwise.
```

EVERY takes a list, denoted by EVERYX, and applies a function, EVERYFN1, to its CAR, e.g., (EVERYFN1 (CAR EVERYX)). If the result of this computation is NIL, then EVERY returns NIL without further evaluations. Otherwise, it applies a second function, EVERYFN2, to generate the new EVERYX. If EVERYFN2 is NIL, EVERY simply computes (CDR EVERYX). Consider the following examples:

EVERYFN1 is NIL, EVERYFN2 is NIL

This is a degenerate form that is equivalent to

```
(MAPCAR everyx 'AND)
```

EVERYFN1 is non-null, EVERYFN2 is NIL

This is the equivalent of applying EVERYFN1 to each element of the list EVERYX. For example, if EVERYFN1 is ATOM, then

```
←(EVERY (LIST 'x 'y 'z) (FUNCTION ATOM))
T
```

EVERYFN1 is NIL, EVERYFN2 is non-null

This form allows a user to select those elements of EVERYX that will be tested by EVERY. Because EVERYFN2 is used to generate each new version of EVERYX, only those elements of EVERY will be tested.

5.4.1 A Definition for EVERY

We might define EVERY as follows:

```
(DEFINEQ
     (every (everyx everyfn1 everyfn2)
          (PROG NIL
          loop
               (COND
                    ((NLISTP everyx)
                         (*
                              If EVERYX is not a list,
                              apply EVERY to a listified
                              form of EVERYX.
                         )
                         (RETURN
                              (EVERY (LIST everyx)
                                           everyfn1)))
                    ((NULL
                         (APPLY* everyfn1
                                 (CAR everyx)
                                 everyx))
                         (*
                              Test the CAR of EVERYX with
                              the specified predicate. If
                              the result is NIL, EVERY
                              fails for this instance of
                              EVERYX.
                         )
                         (RETURN NIL)))
               (SETQ everyx
```

```
                    (COND
                       (everyfn2
                            (*
                                 There is a selector
                                 function! So generate
                                 a new version of
                                 EVERYX.
                            )
                            (APPLY* everyfn2 everyx))
                       (T
                            (*
                                 The default case.
                            )
                            (CDR everyx))))
                    (GO loop))
    ))
```

An alternative form, **NOT-EVERY**, returns the opposite of EVERY if some of the elements of EVERYX do not satisfy EVERYFN1. We might define NOTEVERY as follows:

```
(DEFINEQ
     (notevery (everyx everyfn1 everyfn2)
          (NULL (EVERY everyx everyfn1 everyfn2))
     ))
```

5.4.2 Applications of EVERY

EVERY has numerous applications in INTERLISP programs for testing the consistency of data structures and values of elements of data structures. Consider the following examples:

Test if every member of a list is a number

```
(DEFINEQ
     (numbers? (lst)
          (EVERY lst (FUNCTION NUMBERP))
     ))

←(SETQ lst '(1 -34 2.65 0 0.0 -32.09 2314))
(1 -34 2.65 0 0.0 -32.09 2314)

←(numbers? lst)
T
```

Test if every member of a list is an atom

```
(DEFINEQ
     (atoms? (lst)
          (EVERY lst (FUNCTION ATOM))
     ))

←(SETQ lst '(harding grant coolidge))
(harding grant coolidge)

←(atoms? lst)
T
```

Similar functions may be defined to test any characteristic of a single-level list by substituting the appropriate predicate into the EVERY expression and choosing a suitable name for the function. I believe it makes INTERLISP programs more readable to define new predicate functions in this manner rather than to use EVERY expressions directly in the program code.

5.5 EXISTENTIAL QUANTIFICATION

In predicate logic, a formula P(x) may have the value T if any one of its arguments satisfies the predicate. To assert that some value of X chosen from a suitable domain satisfies P, we place an *existential quantification* symbol in front of the formula. INTERLISP provides a function for testing whether or not one of a list of values satisfies a given predicate.

SOME tests if the application of a given function to each element of a list results in a value of T for some members of that list. It takes the form

```
Function:       SOME
                NOTANY

# Arguments:    3

Arguments:      1) a list, SOMEX
                2) an evaluation function, SOMEFN1
                3) a selector function, SOMEFN2

Value:          The tail of SOMEX, if at least one element
                of SOMEX satisfies SOMEFN1; NIL otherwise.
```

SOME applies SOMEFN1 to the CAR of SOMEX. If that value is non-NIL, SOMEX is returned as the value of SOME. Otherwise, SOMEFN2 is applied to SOMEX. If SOMEFN2 is NIL, SOME uses (CDR SOMEX). That is, it applies SOMEFN1 to each element of SOMEX in succession until one of those elements returns a non-NIL value. The value of SOME is the tail of SOMEX beginning with the element which satisfied SOMEFN1.

Consider the following cases:

SOMEFN1 is NIL, SOMEFN2 is NIL

This is a degenerate case that is equivalent to

```
(MAPCAR somex 'OR)
```

SOMEFN1 is non-null, SOMEFN2 is NIL

This is the equivalent to applying SOMEFN1 to each element of SOMEX. For example, if SOMEFN1 is GREATERP, then

```
←(SOME (LIST 23.0 -45.0 0.0) (FUNCTION GREATERP))
(23.0 -45.0 0.0)
```

5.5.1 A Definition for SOME

We might define SOME as follows:

```
(DEFINEQ
    (some (somex somefn1 somefn2)
        (PROG NIL
        loop
            (COND
                ((NLISTP somex)
                    (*
                        If SOMEX is not a list,
                        apply SOME recursively to a
                        listified form of SOMEX. No
                        need to pass SOMEFN2 since
                        we know there's only one
                        element of the first
                        argument.
                    )
                    (RETURN
                        (SOME (LIST somex)
                        somefn1))))
                ((APPLY* somefn1
                        (CAR somex)
                        somex)
                    (*
                        If this form evaluates to a
                        non-NIL value, we have
                        found at least one element
                        of SOMEX that satisfies the
                        predicate.
                    )
                    (RETURN somex)))
```

```
                (SETQ somex
                        (COND
                        (somefn2
                            (*
                                There is a selector
                                function! So generate
                                the new version of
                                SOMEX.
                            )
                            (APPLY* somefn2 somex))
                        (T
                            (*
                                The default case.
                            )
                            (CDR somex))))
                (GO loop))
    ))
```

An alternative form, **NOTANY**, performs the opposite operation from SOME. That is, if SOME returns NIL, meaning no element satisfied SOMEFN1, then NOTANY would return T.

We might define NOTANY as

```
(DEFINEQ
    (notany (somex somefn1 somefn2)
        (NULL (SOME somex somefn1 somefn2))
    ))
```

6

List Manipulation

Symbolic expressions are also called lists. A list is just a sequence of objects, such as atoms or other lists, enclosed in a pair of parentheses. The essence of programming in INTERLISP is manipulating lists to store information. This chapter discusses the basic list manipulation functions.

6.1 CREATING LISTS

We have discussed the list creation functions in Section 3.2. However, to make this chapter complete, we will summarize these functions for you.

CONS adds a new member to the list by prefixing it to the front of the list which is given as its second argument. It takes the form

```
Function:       CONS
# Arguments:    2
Arguments:      1) any atom or list, EXPRESSION
                2) any list, LST
Value:          A list whose CAR is the value of the first
                argument and whose CDR is the value of the
                second argument.
```

Consider the following example:

```
←(CONS 'x '(y z))
(x y z)
```

If the second argument is not a list but another atom, then CONS produces a form known as a *dotted pair*. This form takes its name from the fact that the pointers to the two atoms occupy the CAR and CDR portions of a CONS cell.

```
←(CONS 'x 'y)
(x . y)
```

LIST creates a new list from its argument which may be atoms or lists. Typically, LIST is used to create a new list from a sequence of atoms. It takes the form

```
Function:       LIST
# Arguments:    1-N
Arguments:      1-N) S-expressions, EXPRESSION[1] ...
                     EXPRESSION[N]
Value:          A list of the values of its arguments.
```

Consider the following example:

```
←(LIST 'apple 'cherry 'lime)
(apple cherry lime)
```

APPEND joins two lists together at their top level. What this means is, figuratively, that if we place the two lists side-by-side and erase the innermost pair of opposing parentheses, we will see the new list take shape. It takes the form

```
Function:       APPEND
# Arguments:    1-N
Arguments:      1-N) lists, LST[1] ... LST[N]
Value:          A list of the S-expressions of the
                individual lists.
```

Consider the following example:

```
←(APPEND '(sherry port) '(riesling pinot-noir))
(sherry port riesling pinot-noir)
```

We can visualize this by placing the lists side-by-side as shown below and erasing the innermost pair of opposing parentheses.

```
( sherry port ) ( riesling pinot-noir )
              ↑ ↑
```

Erase this pair of opposing parentheses to yield

```
(sherry port riesling pinot-noir)
```

6.2 CONCATENATING LISTS

When you append two or more lists together, the result is always a new list. Frequent invocations of APPEND will rapidly consume the available memory forcing the system to spend more of its time in **garbage collecting** the remnants we have left lying around. Moreover, we often want to amend a list without changing its name (as we must do in using APPEND). INTERLISP provides us with several concatenation functions that change the CDR portion of the last cell of all but the last argument when linking the argument lists together. In effect, they "smash" the current value of each last cell's CDR portion and replace it (just as RPLACD does) with a new value—the pointer to the first cell of the next argument list. However, in performing this operation we destroy the integrity of the second argument because it is merged into the first argument.

6.2.1 NCONC: Normal Concatenation

NCONC is the INTERLISP function that performs normal concatenation of two or more lists. Each of its arguments is a list. It modifies the CDR portion of the last cell of each list to point to the first cell of the succeeding argument list. Obviously, this cannot occur for the last argument list, and so it remains unmodified. It takes the form

```
Function:       NCONC
# Arguments:    2...N
Arguments:      lists
Value:          a pointer to the first argument
```

NCONC is a nospread function. Consider the following example:

```
←(SETQ french-wines (LIST 'pinot-noir 'merlot))
(pinot-noir merlot)
←(SETQ german-wines (LIST 'riesling 'sylvaner))
(riesling sylvaner)
←(SETQ wines (APPEND french-wines german-wines))
(pinot-noir merlot riesling sylvaner)
←french-wines
(pinot-noir merlot)
```

```
←german-wines
(riesling sylvaner)

←(SETQ wines french-wines)
(pinot-noir merlot)

←(NCONC wines german-wines)
(pinot-noir merlot riesling sylvaner)

←wines
(pinot-noir merlot riesling sylvaner)
```

where we see that the list WINES has physically been altered by execution of NCONC.

NCONC can take more than two lists as its arguments. In this case, all but the last list will be physically altered by execution of NCONC. NCONC always returns a pointer to the first cell of the first argument as its result.

NCONC may be given NIL as the value of its first argument. Since NIL is treated as both an atom and a list, the following is a valid invocation of NCONC:

```
←(SETQ good-wines NIL)
NIL

←(NCONC good-wines (LIST 'pinot-noir 'merlot))
(pinot-noir merlot)
```

However, NCONC operates somewhat differently when it encounters this situation. We must realize that NCONC deals with pointers to lists rather than the lists themselves. Thus, when it encounters NIL as the value of GOOD-WINES, it does not know that this NIL is the value of GOOD-WINES as opposed to the intrinsic atom/list NIL. The result would be to modify the system atom/list NIL permanently, which would produce future catastrophic results. Thus, NCONC checks to see if its first argument is NIL and, if so, returns a pointer to the second argument.

```
←good-wines
NIL
```

Thus, although the value of the NCONC expression is (pinot-noir merlot), GOOD-WINES has not been changed. NCONC has not changed NIL the "atom" into a list.

Note that the first argument of NCONC cannot be an atom in any case since NCONC is not allowed to change a non-list to a list in order to concatenate it. However, the second argument may be an atom. A variation of NCONC, **NCONC1**, is used to concatenate an atom to the end of a list. It takes the form

```
Function:       NCONC1
# Arguments:    2
Arguments:      1) a list, LST
                2) any S-expression, EXPRESSION
Value:          A list composed of LST concatenated with
                the value of EXPRESSION.
```

We might define NCONC1 as

```
(DEFINEQ
     (NCONC1 (alst an-atom)
          (NCONC alst (LIST an-atom))
  ))
```

Basically, NCONC1 just applies the function LIST to its second argument before performing the concatenation. However, since this operation is frequently performed, NCONC1 is often hardwired to make it more efficient. For example,

```
←(NCONC1 french-wines 'cabernet-sauvignon)
(pinot-noir merlot cabernet-sauvignon)
```

The definition for NCONC in terms of more primitive functions might appear as

```
(DEFINEQ
  (NCONC (list-of-lists)
     (PROG (a-list tail)
          (SETQ a-list
               (OR
                    (SOME list-of-lists (FUNCTION LISTP))
                    (RETURN
                         (CAR (LAST list-of-lists)))))
          (SETQ tail
               (LAST (CAR a-list)))
          (MAPC
               (CDR a-list)
               (FUNCTION
                    (LAMBDA (item)
                         (RPLACD tail item)
                         (SETQ tail (LAST tail)))))
          (RETURN (CAR a-list)))
  ))
```

6.2.2 TCONC: One at a Time Concatenation

When we use NCONC1, INTERLISP must find the end of the list in order to add the new element. Many times we are faced with the situation where we must add multiple elements to the end of the list, but one at a time. As the number of elements to be added grows, NCONC1 rapidly becomes inefficient because it finds the end of the list anew on each invocation.

TCONC solves this problem by remembering where the end of the list is from invocation to invocation. Each time it is called, TCONC inspects a pointer it has created for the list that shows where the end of the list resides in memory. Thus, updating a list can proceed very rapidly when one element at a time is added to its end. It takes the form

```
Function:       TCONC
# Arguments:    2
Arguments:      1) a list having the pointer format,
                POINTER
                2) the element to be added, ELEMENT
Value:          An updated pointer for succeeding
                operations.
```

TCONC can be initialized in two ways:

1. If POINTER is NIL, TCONC creates a POINTER for you.
2. If POINTER has a value, TCONC changes the value of POINTER.

Consider the following examples:

```
←(SETQ wines (TCONC NIL 'merlot))
((merlot) merlot)
```

where the CAR of the list returned is the list that you are building and the CDR is the pointer to the last element added to the list. Thus, you can always determine where you are in building the list if its proper assembly depends on a specific sequence of steps.

When POINTER is (NIL), TCONC will change the value of POINTER. Consider the following example:

```
←(SETQ fruits (LIST NIL))
(NIL)

←(TCONC fruits 'cabernet-sauvignon)
((cabernet-sauvignon) cabernet-sauvignon)

←fruits
((cabernet-sauvignon) cabernet-sauvignon)
```

We usually build a list incrementally when we are repeating a function or sequence of statements several times. One likely candidate is within a PROG. Many times, however, we merely want to repeat one function a fixed number of times where each iteration generates a single element to be added to the list. In this case, we are likely to use the RPTQ or RPT function.

Consider the following example:

```
←(RPTQ 5
     (SETQ list-of-numbers
          (TCONC list-of-numbers rptn)))
((5 4 3 2 1) 1)
```

We recognize that this statement produces a list that appears a lot like the vector produced by the APL index-generation operator. Let us capture it as a function, but produce the list in the proper order.

```
(DEFINEQ
  (index-generation (index)
     (PROG (index-list)
          (SETQ index-list (LIST NIL))
          (SETQ index-list
               (DREVERSE
                    (CAR
                         (RPT index
                              (TCONC index-list RPTN)))))
          (RETURN index-list))
     ))
```

We can define TCONC in terms of the elementary functions as follows

```
(DEFINEQ
  (TCONC (pointer element)
     (PROG (pointer-list)
          (RETURN
               (COND
                    ((NULL pointer)
                         (*
                              POINTER is NIL. Create one
                              with the value of ELEMENT
                              and return it.
                         )
                         (CONS
                              (SETQ pointer-list
                                   (CONS element NIL))
                              pointer-list))
                    ((NLISTP pointer)
```

```
                    (*
                        Generate an error if the
                        pointer is not a list.
                    )
                    (ERROR "Bad Argument-TCONC:"
                           pointer))
                ((NULL (CDR pointer))
                    (*
                        Handle the case of the
                        first-time call, whence
                        there is no end-of-list
                        pointer.
                    )
                    (RPLACA pointer (CONS element NIL))
                    (RPLACD pointer (CAR pointer)))
                (T
                    (*
                        Handle all other cases.
                    )
                    (RPLACD pointer
                       (CDR
                          (RPLACD (CDR pointer)
                                (RPLACD (CONS element)
                                        (CDR pointer)))
                     ))))))
))
```

6.2.3 LCONC: Concatenating Lists

TCONC is used to add elements to the end of a list. Many times, we want to add lists to the end. Again, the problem is that we must always find the end of the list each time we invoke NCONC. **LCONC** (for List Concatenation) maintains a pointer just as TCONC does, but its second argument must always be a list. It takes the form

Function:	LCONC
# Arguments:	2
Arguments:	1) a list having pointer format, POINTER 2) a list, LST
Value:	The pointer list updated with the value of LST.

Consider the following example:

```
←(SETQ wines (LIST NIL))
(NIL)

←(LCONC wines (LIST 'merlot 'sylvaner))
((merlot sylvaner) sylvaner)
```

Note that the CDR portion contains the value of the last list element that is added to the list. This convention is the same as that used for TCONC. Thus, if we had to add both lists and elements to a list, we can call both TCONC and LCONC with the same pointer structure. Given the pointer structure WINES above, consider the following example:

```
←(TCONC wines 'chablis)
(merlot sylvaner chablis) chablis)

←(LCONC wines 'chablis)
bad argument-LCONC
```

because the second argument must be a list.

We can define LCONC in terms of elementary functions as follows

```
(DEFINEQ
  (LCONC (pointer a-list)
   (PROG (pointer-list)
       (SETQ pointer-list (LAST a-list))
       (RETURN
          (COND
             ((NULL a-list)
                   (*
                        If NIL is to be added to the
                        existing list, just return the
                        pointer.
                   )
                   pointer)
             ((NLISTP a-list)
                   (*
                        If the argument to be added is
                        not a list, generate an error.
                   )
                   (ERROR "Bad Argument-LCONC: "
                      a-list))
             ((NULL pointer)
                   (*
                        If POINTER is NIL, create a new
                        pointer with A-LIST as the sole
```

```
                    element, and the last atom of A-
                    LIST as the marker.
               )
               (CONS a-list pointer-list))
          ((NLISTP pointer)
               (*
                    If POINTER is not a list,
                    generate an error.
               )
               (ERROR "Bad Argument-LCONC: "
                      pointer))
          ((NULL (CAR pointer))
               (*
                    Handle the case (NIL).
               )
               (RPLACA
                         (RPLACD pointer pointer-list)
                         a-list))
          (T
               (*
                    Handle all other cases.
               )
               (RPLACD (CDR pointer) a-list)
               (RPLACD pointer pointer-list)))
))
```

6.2.4 ATTACH: Concatenating at the Front

Each of NCONC, TCONC, and LCONC adds elements, whether atoms or lists, to the end of a list. It is often useful to be able to add elements to the front of a list. For example, in maintaining an agenda of tasks to be accomplished, you may want to place the highest priority task at the front of the list.

ATTACH adds an element to the front of the list by doing an RPLACA and RPLACD. Its format is

```
Function:       ATTACH

# Arguments:    2

Arguments:      1) an element, X
                2) a list, LST

Value:          The modified list, LST'.
```

Consider the following example:

```
←(SETQ baryons '(proton muon kaon))
(proton muon kaon)
```

```
←(ATTACH 'omega-minus baryons)
(omega-minus proton muon kaon)

←(ATTACH 'sigma-minus NIL)
(sigma-minus)
```

which is the same as performing (CONS 'sigma-minus NIL).

```
←(ATTACH (LIST 'tau-minus) baryons)
((tau-minus) omega-minus proton muon kaon)
```

ATTACH performs destructive modification of the list. The resulting list remains EQ to LST.

If the second argument is not a list, INTERLISP generates an error message ARG NOT LIST.

We might define ATTACH as follows:

```
(DEFINEQ
     (attach (x lst)
          (COND
               ((LISTP lst)
                    (RPLACA
                         (RPLACD lst
                                 (CONS (CAR lst)
                                       (CDR lst)))
                         x))
               ((NULL lst)
                    (*
                         The second argument must be a
                         list.
                    )
                    (ERROR "ARG NOT LIST" x)))
     ))
```

6.2.5 Variations on Concatenation

The CONCatenation functions use an extra CONS cell to keep track of the pointers to the list elements. INTERLISP provides two functions that avoid the overhead of the extra CONS cell: **DOCOLLECT** and **ENDCOLLECT**. They take the form

```
Function:       DOCOLLECT
                ENDCOLLECT

# Arguments:    2
```

```
Arguments:      1) an item, ITEM
                2) a list, LST

Value:          A list with the item inserted.
```

Consider the following examples:

```
←(SETQ particles '(neutrino electron W-boson))
(neutrino electron W-boson)

←(DOCOLLECT 'neutron particles)
(neutron electron W-boson)

←particles
(neutrino neutron electron W-boson)
```

Caution must be exercised in using DOCOLLECT because you may cause the machine to enter into an infinite loop that may only be exited by rebooting. Consider the example

```
←(SETQ particles NIL)
NIL

←(DOCOLLECT 'positron particles)
```

At this point, the machine begins printing an endless list as follows:

```
(positron positron positron positron positron positron
positron positron positron ... ad infinitum

←(SETQ particles (LIST 'neutrino 'electron 'W-boson))
(neutrino electron W-boson)

←(ENDCOLLECT particles 'tau-minus)
(electron W-boson)

←particles
(neutrino . tau-minus)

←(ENDCOLLECT particles (LIST 'J-particle 'omega-zero))
(electron W-boson)

←particles
(neutrino J-particle omega-zero)

←(ENDCOLLECT NIL 'electron)
electron

←(SETQ particles '(electron proton))
(electron proton)
```

```
←(ENDCOLLECT particles NIL)
(proton)

←particles
(electron)
```

As we see, DOCOLLECT adds an item at the beginning of the CDR of the list, whereas ENDCOLLECT replaces the CDR of the list. In general, DOCOLLECT maintains the list that it is building as a circular list. When you are ready to add the last item, you should use ENDCOLLECT, which returns a non-circular list. Thus, DOCOLLECT and ENDCOLLECT are meant to be complementary functions.

Implementing DOCOLLECT

DOCOLLECT may be implemented in terms of RPLACD as follows:

```
(DEFINEQ
     (docollect (an-item a-list)
          (COND
               ((NLISTP a-list)
                    (RPLACD (SETQ a-list (LIST item))
                            a-list))
               (T
                    (CDR (RPLACD a-list
                                 (CONS item a-list)))))
     ))
```

Implementing ENDCOLLECT

ENDCOLLECT may be implemented using RPLACD as follows:

```
(DEFINEQ
     (endcollect (item a-list)
          (COND
               ((NULL item) a-list)
               (T
                    (PROG1
                         (CDR item)
                         (RPLACD item a-list))))
     ))
```

Using DOCOLLECT in MAPCAR

The IRM [irm78] suggests that MAPCAR may be defined, using DOCOLLECT and ENDCOLLECT, as follows:

```
(DEFINEQ
    (mapcar (a-list a-function)
   (PROG (value)
```

```
    loop
         (COND
            ((NLISTP a-list)
                 (RETURN (ENDCOLLECT value))))
         (SETQ value
              (DOCOLLECT
                   (APPLY* a-function (CAR a-list))
                   value))
          (SETQ a-list (CDR a-list))
          (GO loop))
))
```

6.3 SUBLIST EXTRACTION

In Chapter 2, we saw that one way to take a list apart was to use the functions CAR and CDR. These work, respectively, on the head and the tail of a list. In addition, INTERLISP provides several functions that can operate on interior components of a list.

6.3.1 Extracting the Last Element

LAST allows you to retrieve the last node in a list, e.g., the contents of the last list cell. LAST takes the form

```
Function:       LAST
# Arguments:    1
Arguments:      1) a list, LST
Value:          A list whose element is the last node in
                LST; otherwise, NIL.
```

If its argument is not a list, LAST returns NIL:

```
←(LAST 'rhomboid)
NIL
```

Otherwise, it returns the contents of the last cell of LST as a list. There are two possible cases:

1. The last cell was appended (NIL in the CDR part). For example,

```
←(LAST '(parallel quadrilateral rhomboid))
(rhomboid)
```

2. The last cell was CONSed to the list. For example,

```
←(LAST '(parallel triangle quadrilateral . rhomboid))
(quadrilateral . rhomboid)
```

LAST is useful when you must obtain the last entry of a list. If the list is longer than four elements, you cannot use one of the CAR ... CDR combinations to retrieve the element. Using LAST, you do not need to know the length of the list to retrieve the last element.

We might define LAST as follows:

```
(DEFINEQ
     (last (lst)
          (PROG (xprev)
                (SETQ xprev NIL)
          loop
                (COND
                   ((NLISTP lst)
                        (RETURN xprev)))
                (SETQ xprev lst)
                (SETQ lst (CDR lst))
                (GO loop))
     ))
```

Note that XPREV always holds the previous element of X. Thus, when we determine that X is no longer a list, the previous element must be a list. Hence, it is returned as the last element of X. On the first pass through the code, if X is not a list, then NIL will be returned. See [knut68] for a detailed discussion of list manipulation algorithms.

6.3.2 Extracting the Tailing N Elements

NLEFT allows you to extract the rightmost N elements of a list where N is greater than the number of elements in a specified tail of the list. Its format is

```
Function:       NLEFT

# Arguments:    3

Arguments:      1) a list, LST
                2) extension, N
                3) the tail, TAIL

Value:          A list with N more elements than the tail.
```

Consider the following examples:

1. TAIL is NIL (the usual case):

```
←(SETQ awards '(tony oscar emmy coty))
(tony oscar emmy coty)

←(NLEFT awards 2)
(emmy coty)
```

2. TAIL is non-NIL:

```
←(SETQ tail (CDDDR awards))
(coty)

←(NLEFT awards 1 tail)
(emmy coty)
```

where the length of the tail is 1, and we are asking for a list whose length is 1 greater than the length of the tail.

```
←(NLEFT awards 4 tail)
NIL
```

because there is no list that may be extracted from the first argument which meets the specified criterion, namely having a length 4 elements greater than the tail.

If LST is not a list and is equivalent to TAIL, NLEFT returns NIL, except in the case where N is zero:

```
←(NLEFT (LAST awards) 1 tail)
NIL

←(NLEFT (LAST awards) 0 tail)
(coty)
```

You may use NLEFT to work backwards through a list by setting TAIL to (LAST lst). Then, by calling NLEFT repreatedly with different values of N and using CAR to strip off the head of the resulting list, you access the list in reverse order. When the result is NIL, you know that you have reached the end of the list.

We might define NLEFT as follows:

```
(DEFINEQ
     (nleft (lst n tail)
```

```
(PROG (x)
      (SETQ x lst)
loop
      (COND
          ((ZEROP n)
               (*
                    N equal to 0 is equivalent
                    to LAST.
               )
               (GO loop1))
          ((OR
               (EQ x tail)
               (NLISTP x))
               (*
                    Return NIL if:
                    1.   The first argument is
                         not a list.
                    2.   It is equal to the
                         thing that is looked
                         for.
               )
               (RETURN NIL)))
      (SETQ x (CDR x))
      (SUB1VAR n)
      (GO loop)
loop1
      (COND
          ((OR
               (EQ x tail)
               (NLISTP x))
               (*
                    Return 0 if N is 0.
               )
               (RETURN lst)))
       (SETQ x (CDR x))
       (SETQ lst (CDR lst))
       (GO loop1))
))
```

Note that if N is zero and TAIL is NIL, then NLEFT is equivalent to LAST. If N is zero and TAIL is non-NIL, then NLEFT operates like MEMBER.

6.3.3 Extracting the Last N Elements

LASTN extracts the last N elements of a list. It takes the form

```
Function:       LASTN
# Arguments:    2
Arguments:      1) a list, LST
                2) an index, N
Value:          An S-expression consisting of the initial
                and final segments of a list.
```

Assume LST has length equal to L elements. LASTN "breaks" a list at the Nth element. It returns an S-expression equivalent to

```
(CONS initial . final)
```

where

Initial The first through L-N-1 elements of the list
Final The Nth through L elements of the list

If LST does not contain N elements (i.e., L less than N), LASTN returns NIL.

Consider the following examples:

```
←(SETQ lst '(the lazy fox jumped over the brown dog))
(the lazy fox jumped over the brown dog)
←(LASTN lst 5)
((the lazy fox) jumped over the brown dog)
←(LASTN lst 10)
NIL
←(LASTN lst 8)
(NIL the lazy fox jumped over the brown dog)
```

LASTN provides a convenient mechanism for decomposing complex list structures. The CAR of its result always returns the initial segment while the CDR returns the final segment. Let us assume the following complex structure:

```
(class conceptualization properties constituents)
```

which might be the syntax node structure in a natural language parser.

We know that the list has a fixed structure but that the type of individual elements (except for the first) may be atoms or lists. We can defined the following functions:

```
(DEFINEQ
     (syntax:class (node)
          (car node)))
(DEFINEQ
     (syntax:concept (node)
          (CADR (LASTN node 3))))
(DEFINEQ
     (syntax:properties (node)
          (CADR (LASTN node 2))))
(DEFINEQ
     (syntax:constituents (node)
          (CADR (LASTN node 1))))
```

Note that we can extract an individual element by taking the CADR of LASTN of the original list where the element is the Nth element from the tail of the list.

Although I have demonstrated this approach for a list of four elements, you can see that it is easily applicable to a list of N elements provided N is fixed.

A Definition for LASTN

We might define LASTN as follows:

```
(DEFINEQ
     (lastn (lst n)
          (PROG (xlst ylst)
                (SETQ xlst (NTH lst n))
                (COND
                    ((NLISTP lst)
                         (*
                              If LST is not a list.
                         )
                         (RETURN NIL))
                    ((NULL xlst)
                         (*
                              If N is greater than the
                              length of LST.
                         )
                         (RETURN xlst)))
          loop
                (SETQ xlst (CDR xlst))
                (COND
                    ((NULL xlst)
                         (RETURN
                              (CONS ylst lst))))
```

```
                (SETQ ylst
                      (NCONC1 ylst (CAR lst)))
                (SETQ lst (CDR lst))
                (GO loop))
    ))
```

6.3.4 Extracting From the Nth Element

Given a list of K elements, how do you extract from the Nth element (where N is less than K)? If N is 1, 2, or 3, you can use some combination of CARs and CDRs to retrieve it. You may use LAST if N equals K. Otherwise a complex function might be required with substantial testing for the length and end of the list for different cases. INTERLISP provides **NTH** to extract the tail of a list beginning with the Nth element where the length of the list is unknown. It takes the form

Function:	NTH
# Arguments:	2
Arguments:	1) an S-expression, EXPRESSION 2) an index, N
Value:	The tail of the list beginning with the Nth element.

Let us consider several cases, using the list

```
←(SETQ games
       (LIST 'poker 'gin-rummy 'hearts 'bridge
       'canasta))
(poker gin-rummy hearts bridge canasta)
```

1. If N is 0, the value is (CONS NIL EXPRESSION).

```
←(NTH games 0)
(NIL poker gin-rummy hearts bridge canasta)
```

2. If N is 1, the value is EXPRESSION.

```
←(NTH games 1)
(poker gin-rummy hearts bridge canasta)
```

3. If N is 2, the value is (CDR EXPRESSION).

```
←(NTH games 2)
(gin-rummy hearts bridge canasta)
```

4. If (LENGTH EXPRESSION) is less than N, the value is NIL.

```
←(NTH games 7)
NIL
```

5. Otherwise, the tail of EXPRESSION beginning with the NTH element is returned.

```
←(NTH games 3)
(hearts bridge canasta)
```

We might define NTH as follows:

```
(DEFINEQ
     (nth (expression n)
          (COND
               ((IGREATERP 1 N)
                    (*
                         If N is less than or equal to
                         zero.
                    )
                    (CONS NIL expression)))
          (PROG NIL
          loop
                (COND
                    ((EQUAL n 1)
                         (RETURN expression))
                    ((NLISTP expression)
                         (RETURN NIL)))
                (SETQ expression (CDR expression))
                (SETQ n (SUB1 n))
                (GO loop))
     ))
```

6.4 COPYING AND REVERSING LISTS

INTERLISP duplicates pointers to lists rather than duplicating the lists themselves in many functions. For example,

```
←(SETQ computers
        (LIST 'atari-1200 'apple-IIe 'TRS-80))
(atari-1200 apple-IIe TRS-80)
←(SETQ microcomputers computers)
(atari-1200 apple-IIe TRS-80)
```

creates an additional pointer to the list which is the value of COMPUTERS. We can verify this by checking to see if MICROCOMPUTERS and COMPUTERS are EQ.

```
←(EQ microcomputers computers)
T
```

However, if we subsequently modify COMPUTERS, then the value of MICROCOMPUTERS is modified as well.

```
←(SETQ computers (APPEND computers 'iAPX-286))
(atari-1200 apple-IIe TRS-80 iAPX-286)

←microcomputers
(atari-1200 apple-IIe TRS-80 iAPX-286)
```

However, if we wish to modify COMPUTERS without modifying MICROCOMPUTERS, then we must make a copy of COMPUTERS. This section describes functions for copying lists.

6.4.1 Copying List Elements

COPY makes a copy of the list which is its argument. It returns the new list as its value. COPY duplicates elements of its arguments down to the non-list level. However, if some of its elements are strings or arrays, the new list will contain these same strings or arrays (via pointers to them). It takes the form

```
Function:       COPY
# Arguments:    1
Argument:       1) an S-expression to be copied,
                   EXPRESSION
Value:          A copy (with new storage assigned) of its
                argument.
```

Consider the following examples:

```
←(SETQ countries '(france spain denmark norway))
(france spain denmark norway)

←(SETQ same-countries (COPY countries))
(france spain denmark norway)

←(EQ countries same-countries)
NIL

←(EQUAL countries same-countries)
T
```

demonstrates that while the lists are equivalent, they are not the same data structure.

```
←(SETQ some-sf-writers '("Asimov" "Dick"
"Saberhagen"))
("Asimov" "Dick" "Saberhagen")
←(SETQ good-sf-writers (COPY some-sf-writers))
("Asimov" "Dick" "Saberhagen")
←(EQ some-sf-writers good-sf-writers)
NIL
←(EQUAL some-sf-writers good-sf-writers)
T
←(SETQ seed 'tamarind)
tamarind
←(SETQ new-seed (COPY seed))
tamarind
←(EQ seed new-seed)
T
```

If you just want to copy the top level of the list, you may use (APPEND expression).

We might define COPY as follows:

```
(DEFINEQ
     (copy (expression)
          (COND
               ((NLISTP expression)
                    (*
                         If EXPRESSION is not a list,
                         just return the value of
                         EXPRESSION.
                    )
                    expression))
          (PROG (xlst ylst)
               (*
                    Copy the first element of expression.
               )
               (SETQ ylst
                    (LIST (COPY (CAR expression))))
               (SETQ xlst ylst)
          loop
               (SETQ expression (CDR expression))
```

```
                    (COND
                        ((NLISTP expression)
                            (*
                                Condition is satisfied when
                                we reach the end of
                                expression, since (NLISTP
                                NIL) is T.
                            )
                            (RPLACD ylst (COPY expression))
                            (RETURN xlst)))
                    (SETQ ylst
                          (CDR
                            (RPLACD ylst
                                    (RPLACD
                                        (CONS
                                            (COPY
                                                (CAR
                                                expression))
                                            ylst)))))
                    (GO loop))
        ))
```

6.4.2 Copying All List Elements

COPY will not duplicate non-list elements when it copies an expression (see "seed" example above). Rather, it creates pointers to non-list elements such as arrays, strings, etc. **COPYALL** duplicates every element of a list including atoms, arrays, and strings when you must have new copies of each element of the list. It takes the form

Function:	COPYALL HCOPYALL
# Arguments:	1
Argument:	1) an S-expression, EXPRESSION
Value:	A new copy of EXPRESSION including all non-list data structures.

Consider the following examples:

```
←(SETQ x1 "Merry Christmas")
"Merry Christmas"

←(SETQ x2 (COPYALL x1))
"Merry Christmas"
```

```
←(EQ x1 x2)
NIL

←(EQUAL x1 x2)
T

←x2
"Merry Christmas"

←(SETQ y (ARRAY 3))
{ARRAYP}#542224

←(SETA y 1 100)
100

←(SETA y 2 200)
200

←(SETA y 3 300)
300

←(SETQ z (COPYALL y))
{ARRAYP}#542231

←(ELT z 2)
200
```

A variation, **HCOPYALL**, copies data structures that contain circular pointers. For example,

```
←(SETQ rare-gases
       (LIST 'helium 'krypton 'argon 'xenon 'radon))
(helium krypton argon xenon radon)
```

Now let us create a circular list:

```
←(RPLACD (LAST rare-gases) rare-gases)
```

The result of executing this function at the top level of INTERLISP is a repeating list of the form

```
(helium krypton argon xenon radon helium krypton argon
xenon radon ...)
```

To terminate the printing, you must interrupt via CTRL-D to force a reset of the top level of INTERLISP.

If we attempt to apply COPYALL to this circular list, we obtain

```
←(SETQ inert-gases (COPYALL rare-gases))
```

This function never succeeds because COPYALL attempts to build up a list by successive references to elements of RARE-GASES. The ultimate result is to exhaust your virtual memory and cause INTERLISP to crash.

However, if we apply HCOPYALL, we obtain

```
←(SETQ inert-gases (HCOPYALL rare-gases))
(helium krypton argon xenon radon . [1])
```

where the strange notation indicates that INERT-GASES is circular just like RARE-GASES.

6.4.3 Copying with Reversal

REVERSE allows you to copy a list while reversing the order of its elements. It takes one argument, the list to be reversed, and returns the reversed list. It takes the form

Function:	REVERSE DREVERSE
# Arguments:	1
Argument:	1) an S-expression, EXPRESSION
Value:	A copy of EXPRESSION with all top-level elements reversed in sequence.

Consider the following examples:

```
←(REVERSE 'huckleberry)
NIL
```

because it does not operate upon non-lists.

```
←(REVERSE (LIST 'orange 'blue 'magenta 'yellow))
(yellow magenta blue orange)
```

REVERSE will only reverse the top-level elements in a list. For example,

```
←(SETQ errors
       (LIST '(my blunder)
             '(his oversight)
             '(their negligence)))
((my blunder) (his oversight) (their negligence))
←(REVERSE errors)
((their negligence) (his oversight) (my blunder))
```

You may wish to define a function, REVERSEALL, that reverses a list at all levels.

An alternative version, **DREVERSE**, destroys the original list and sets the reversed list as its value. It does not use any additional storage.

REVERSE is useful in many situations, but particularly in those programs where you are simulating a stack or an agenda.

Suppose you need to read a sentence into your program. Each sentence is terminated by a ! or a ? which indicates the type of processing to be applied to the sentence. We want to define a function GET-SENTENCE which reads atoms from the your terminal until it detects one of those two terminators. Its definition looks like this

```
(DEFINEQ
     (get-sentence NIL
          (PROG (sentence)
          loop
                (SETQ sentence
                      (CONS (RATOM) sentence))
                (COND
                    ((MEMBER (CAR sentence) '(! ?))
                         (RETURN
                              (CONS (REVERSE sentence)
                                    NIL))))
                (GO loop))
     ))
```

We might define REVERSE as follows:

```
(DEFINEQ
     (reverse (expression)
          (PROG (lst)
           loop
                (COND
                    ((NLISTP expression)
                         (RETURN lst)))
                (SETQ lst
                      (CONS (CAR expression) lst))
                (SETQ expression (CDR expression))
                (GO loop))
     ))
```

6.4.4 Removing Elements from a List

To remove an element from a list, we must excise it from the front, back, or within the list. Removing an element from the front of the list is very simple—we

just return the CDR of the list. Removing an element from the rear is also simple because we can use RPLACA to assign NIL to the last element of the list. Removing an element from within a list is more difficult because we must search for the element and then delete it while adjusting the pointers from the surrounding elements. INTERLISP provides **REMOVE** to delete elements within a list. It takes the form

```
Function:       REMOVE
                DREMOVE

# Arguments:    2

Arguments:      1) an atom, X
                2) an S-expression, LST

Value:          A copy of LST with all top level elements
                equal to X removed from it.
```

Consider the following examples:

```
←(REMOVE 'a '(c r a w d a d d y))
(c r w d d d y)
REMOVE can delete NIL from a list:

←plaintext
(E P L U R I B U S U N U M)

←(RPLACA (LAST plaintext))
(NIL)

←plaintext
(E P L U R I B U S U N U NIL)

←(REMOVE NIL plaintext)
(E P L U R I B U S U N U)
```

Note that REMOVE only removes the top-level elements equal to X. For example,

```
←(REMOVE 'sheila '(sheila (sheila)))
((sheila))
```

And removing the only element of a list makes that list become NIL. For example,

```
←(REMOVE 'apple-pie '(apple-pie))
NIL
```

An alternative form, **DREMOVE**, uses EQ instead of EQUAL to delete X. Moreover, it actually modifies LST rather than returning a copy with X deleted. However, DREMOVE cannot change a list to NIL. For example,

```
←(SETQ dessert '(apple-pie))
(apple-pie)

←(DREMOVE 'apple-pie dessert)
NIL

←dessert
(apple-pie)
```

This execution of DREMOVE returns NIL, but it does not perform any CONSes. The value of DESSERT remains (apple-pie), because there is no way to change a list into a non-list.

We might define REMOVE as follows:

```
(DEFINEQ
     (remove (x lst)
          (COND
               ((NLISTP lst) NIL)
               ((EQUAL x (CAR lst))
                    (REMOVE x (CDR lst)))
               (T
                    (CONS (CAR lst)
                          (REMOVE x (CDR lst)))))
     ))
```

6.5 MODIFYING LISTS BY SUBSTITUTION

Lists are a generalized structure for representing information about problems. Often, the ordering of elements within a list has significant import to the interpretation of the list within a program. As program execution progresses, you may want to modify the structure of a list to replace old information with new values. One way is to modify the list itself by substituting new values for those that already exist in the list.

6.5.1 A General Substitution Function: SUBST

SUBST is the general INTERLISP substitution function. Its format is

```
Function:      SUBST
               DSUBST
               LSUBST
```

```
# Arguments:    3
Arguments:      1) an S-expression, NEW
                2) an atom, OLD
                3) a list, LST
Value:          The new value of the list after
                substitution.
```

SUBST evaluates its arguments. It performs a one-for-one substitution of the value of NEW for the value of OLD for all occurrences of OLD in LST when

1. OLD is EQUAL to the CAR of some sublist of LST,
2. When OLD is atomic and not NIL,
3. When OLD is atomic, not NIL, and EQ to the CDR of some sublist of LST.

For example, suppose we have a mathematical formula to evaluate that has placeholders for arguments. We need to substitute the variable names into the formula and then evaluate it. Consider the following example:

```
←(SETQ formula
       '(SQRT (PLUS (TIMES $x $x) (TIMES $y $y))))
(SQRT (PLUS (TIMES $x $x) (TIMES $y $y)))
←(SETQ length 10)
10
←(SETQ height 20)
20
←(EVAL (SUBST 'length '$x
              (SUBST 'height '$y formula)))
22.36
```

How was this accomplished? First, the substitution is performed to yield the S-expression to be evaluated. This is just a list that is given to EVAL to interpret. EVAL returns the value. The intermediate steps in the evaluation are

```
(SUBST 'height '$y formula) yields a list of the form
(SQRT (PLUS (TIMES $x $x) (TIMES height height))).
```

This is then used by (SUBST 'length '$x ...) to produce the list

```
(SQRT (PLUS (TIMES length length) (TIMES height
height))).
```

This list is evaluated by EVAL. Note that LENGTH and HEIGHT in the final list are the names of variables that have values, as defined above, which are evaluated by EVAL to produce the result.

The value of SUBST is a new list containing the appropriate changes to LST. If NEW is a list, its value is copied into the new list at each occurrence of a substitution.

DSUBST is an alternative form of SUBST that does not copy LST but changes its structure. However, it does use a new copy of NEW.

A Definition for SUBST
We might define SUBST as follows:

```
(DEFINEQ
     (subst (new old lst)
          (COND
               ((NULL lst) NIL)
               ((NLISTP lst)
                    (COND
                         ((EQ old lst)
                              (COND
                                   ((NLISTP new) new)
                                   (T (COPY new))))
                         (T lst)))
               (T
                    (CONS
                         (COND
                              ((COND
                                   ((LITATOM old)
                                        (EQ old (CAR
                                        lst)))
                                   (COND
                                        ((NLISTP new)
                                        new)
                                        (T (COPY new))))
                              (T
                                 (SUBST new old (CAR
                                 lst))))
                         (SUBST new old (CDR lst)))))))
```

6.5.2 Substituting by Segments: LSUBST

LSUBST is similar to SUBST except that it substitutes "NEW" segments for OLD. That is, if the value of NEW is a list, when NEW is substituted for OLD in some LST, the elements of NEW become individual elements of LST. For example,

```
←(SETQ sentence
       '(the boy sees the girl with the telescope))
(the boy sees the girl with the telescope)

←(LSUBST '(loves) 'sees sentence)
(the boy loves the girl with the telescope)

←(LSUBST '(the old man) 'boy sentence)
(the old man sees the girl with the telescope)

←(LSUBST '(the green engine) 'john 'john)
(the green engine)
```

We might define LSUBST as follows:

```
(DEFINEQ
     (lsubst (new old lst)
          (COND
               ((NULL lst) NIL)
               ((NLISTP lst)
                    (COND
                         ((EQ old lst) new)
                         (T lst)))
               ((EQUAL old (CAR lst))
                    (NCONC (COPY new)
                           (LSUBST new old (CDR lst))))
               (T
                    (CONS
                         (LSUBST new old (CAR lst))
                         (LSUBST new old (CDR lst)))))
     ))
```

If LST is empty, no substitution can be performed, so LSUBST merely returns NIL. If LST is not a list, and OLD is equal to LST, the value of NEW is substituted for LST. Thus, you may dissect a list component by component and replace individual elements.

6.5.3 Substituting by Association: SUBLIS

Given an expression consisting of many atoms composed into a complex structure, you may want to perform wholesale substitution from one or more atoms in the expression. **SUBLIS** allows you to substitute for multiple atoms with one function invocation. It takes the form

```
Function:      SUBLIS
# Arguments:   3
```

```
Arguments:      1) an association list, ALST
                2) an S-expression, EXPRESSION
                3) a structure flag, FLAG

Value:          A new expression with the appropriate
                substitutions made according to ALST.
```

ALST is a list of pairs having the form

```
((⟨atom1⟩ . ⟨newatom1⟩) ... (⟨atomN⟩ . ⟨newatomN⟩))
```

SUBLIS substitutes ⟨newatom[i]⟩ for each ⟨atom[i]⟩ that is found in the expression. For example,

```
←(SUBLIS '((A B)) '(A B A B A B))
((B) B (B) B (B) B)
```

A new structure may be created if needed, or if FLAG is T. If FLAG is NIL, and there are no substitutions made, the value returned is EXPRESSION.

SUBLIS and SUBPAIR (see below) substitute the identical structure into EXPRESSION (unless FLAG is T) while SUBST and LSUBST substitute copies. Consider the following example:

```
←(SETQ fruits '(papaya guava))
(papaya guava)
←(SETQ cheeses '(edam gouda camembert))
(edam gouda camembert)
←(DSUBLIS (LIST (CONS 'edam fruits)) cheeses)
((PAPAYA GUAVA) GOUDA CAMEMBERT)
←(DSUBLIS (LIST (CONS 'gouda fruits)) cheeses T)
((PAPAYA GUAVA) (PAPAYA GUAVA) CAMEMBERT)
←(EQ (CAR cheeses) fruits)
T
←(EQ (CADR cheeses) fruits)
NIL
```

An alternative form of SUBLIS, **DSUBLIS**, modifies EXPRESSION rather than copying it.

We might define SUBLIS as follows:

```
(DEFINEQ
     (sublis (alst expression flag)
          (COND
```

```
                (alst
                    (SUBPR expression alst))
                (flag
                    (*
                        No substitutions; create new
                        copy.
                    )
                    (COPY expression))
                (T  expression))
    ))
```

SUBPR is defined in the next section. SUBLIS may be used to implement a simple substitution cipher system. Let the association list entries have the form

```
((plaintext letter) (ciphertext letter))
```

The association list has 26 entries (assuming no numbers or special characters). Consider the following key:

Plaintext:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Ciphertext:

D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

which is just the letters of the alphabet circulated by three characters.

A function to encipher plaintext is just

```
(DEFINEQ
    (encipher (text key)
        (SUBLIS key text T)
    ))
```

Deciphering is done, of course, by reversing all of the entries in the key and applying it to the enciphered text.

We may demonstrate this technique by applying it to several examples. First, let us define the key as follows:

```
←(SETQ KEY
    (LIST '(A . D) '(B . E) '(C . F) '(D . G) '(E . H)
          '(F . I) '(G . J) '(H . K) '(I . L) '(J . M)
          '(K . N) '(L . O) '(M . P) '(N . Q) '(O . R)
          '(P . S) '(Q . T) '(R . U) '(S . V) '(T . W)
          '(U . X) '(V . Y) '(W . Z) '(X . A) '(Y . B)
          '(Z . C)))
((A . D) (B . E) (C . F) ...)
```

```
←(SETQ plaintext '(E P L U R I B U S U N U M))
(E P L U R I B U S U N U M)

←(SETQ ciphertext (ENCIPHER plaintext key))
(H S O X U L E X V X Q X P)

←(SETQ plaintext2
       '(T W A S T H E N I G H T B E F O R E X M A S))
(T W A S T H E N I G H T B E F O R E X M A S)

←(SETQ ciphertext2 (ENCIPHER plaintext2 key))
(W Z D V W K H Q L J K W E H I R U H A P D V)
```

You may reverse the entries in the key using the following function

```
(DEFINEQ
     (reverse-key (key)
          (MAPCAR key (FUNCTION REVERSE))
     ))
```

To decipher the text derived above, you may use the expression

```
←(ENCIPHER ciphertext (REVERSE-KEY key))
(E P L U R I B U S U N U M)

←(ENCIPHER ciphertext2 (REVERSE-KEY key))
( T W A S T H E N I G H T B E F O R E X M A S)
```

Note that each entry must be in the form of a CONS list. If each entry was a list of the form (A B), for example, then the resulting list produced by ENCIPHER would appear as

```
((H) (S) (O) (X) (U) (L) (E) (X) (V) (X) (Q) (X) (P))
```

6.5.4 Substituting by Pairing: SUBPAIR

SUBPAIR operates like SUBLIS except that the old and new values are contained in separate lists. It takes the form

```
Function:       SUBPAIR

# Arguments:    4

Arguments:      1) a key list, OLD
                2) a replacement list, NEW
                3) an S-expression, EXPRESSION
                4) a structure flag, FLAG

Value:          The modified expression.
```

For each atom in OLD, SUBPAIR substitutes the corresponding atom in NEW for every occurrence in EXPRESSION. There are several cases:

1. If OLD is an atom, the entire list NEW is substituted for it. For example, in creating a file header, you may specify a form for the File Package command as

```
(FNS * ⟨filefns⟩)
```

where ⟨filefns⟩ is an atom whose value is a list of functions in the file. You may substitute the value of ⟨filefns⟩ for the atom in the form using

```
←(SETQ fnsform (LIST 'FNS '* 'MYFNS))
(FNS * MYFNS)

←(SUBPAIR 'MYFNS MYFNS FNSFORM T)
(FNS * MYFNS1 MYFNS2 MYFNS3 ...)
```

and you may remove the asterisk (*) via DREMOVE (see Section 6.4.4).

2. If (LENGTH OLD) is less than (LENGTH NEW) and OLD ends in an atom other than NIL, the remaining elements of NEW are substituted for the last element of OLD. For example,

```
←(SUBPAIR '(a b . c) '(d e f g) '(x y c w))
(x y (f g) w)
```

because A is matched with D, B is matched with E, and (F G) is matched with C.

3. If (LENGTH NEW) is less than (LENGTH OLD), then NIL is matched with each of the remaining atoms on OLD, For example,

```
←(SUBPAIR '(a b c d) '(x y z) '(c a b b y))
(z x y y y)

←(SUBPAIR '(a b c d) '(x y z) '(d a d d y))
(NIL x NIL NIL y)
```

As with SUBLIS, a new structure is created only if needed or if FLAG is T. That is, if FLAG is NIL, and no substitutions are performed, then the expression returned is EXPRESSION.

We may use SUBPAIR instead of SUBLIS in our cryptographic example given above. We define two key lists as follows:

```
←(SETQ oldkey '(a b c d ...))
(a b c d ...)
←(SETQ newkey '(d e f g ...))
(d e f g ...)
```

We modify the definition of ENCIPHER as follows:

```
(DEFINEQ
     (encipher (oldkey newkey text)
          (SUBPAIR oldkey newkey text T)
     ))
```

Then, we may apply it to an example:

```
←(SETQ ciphertext (encipher oldkey newkey plaintext))
(H S O X U E X V X Q X P)
```

Note that we need only reverse the order of the keys in this definition of ENCIPHER in order to decipher the text.

```
←(SETQ newplaintext (encipher newkey oldkey ciphertext))
(E P L U R I B U S U N U M)
```

We might define SUBPAIR as follows:

```
(DEFINEQ
     (subpair (old new expression flag)
          (COND
               (old
                    (SUBPR expression
                           old
                           (RPLACA '((D E F)) new)))
               (flag
                    (COPY expression))
               (T
                    expression))
     ))
```

Both SUBLIS and SUBPAIR make use of **SUBPR** to perform the hard work of substitution. SUBPR might be defined as follows:

```
(DEFINEQ
     (subpr (expression lst1 lst2)
          (PROG (index dlst alst)
```

```
                (COND
                    ((NLISTP expression)
                        (COND
                            ((NULL lst2)
                                (*
                                    Called from
                                    SUBLIS.
                                    LST2 is NIL.
                                )
                                (GO loop2)))
                            (SETQ lst2 (CAR lst2))
                            (SETQ index 1)
                            (GO loop))
                    ((SETQ dlst (CDR expression))
                       (SETQ dlst
                          (subpr (CAR expression)
                                 lst1
                                 lst2))))
                (SETQ alst
                      (subpr (CAR expression)
                             lst1
                             lst))
                (RETURN
                    (COND
                        ((OR flag
                             (NEQ alst (CAR expression))
                             (NEQ dlst (CDR
                             expression)))
                             (CONS alst dlst))
                        (T expression)))
loop
                (*
                    Searching for LST1 element in
                    expression.
                )
                (COND
                    ((NULL lst1)
                        (RETURN expression))
                    ((NLISTP lst1)
                        (COND
                            ((EQ expression lst1)
                                (GO loop1)))
                        (RETURN expression))
                    ((EQ expression (CAR lst1))
                        (GO loop1)))
```

```
            (SETQ index (ADD1 index))
            (SETQ lst1 (CDR lst1))
            (GO loop)
      loop1
            (*
                At this point, we have found an
                element in expression from LST1; now
                find corresponding element in LST2.
            )
            (COND
                ((EQ index 1)
                     (SETQ lst2
                          (COND
                              ((NLISTP lst1) lst2)
                              (T   (CAR lst2))))
                     (RETURN
                          (COND
                              (flag     (COPY lst2))
                              (T        lst2)))))
            (SETQ index (SUB1 index))
            (SETQ lst2 (CDR lst2))
            (GO loop1)
      loop2
            (COND
                ((EQ (CAAR lst1) expression)
                     (RETURN
                          (COND
                              (flag (COPY (CDAR
                              lst1)))
                              (T (CDAR lst1)))))
                ((NULL (SETQ lst1 (CDR lst1)))
                     (RETURN expression)))
            (GO loop2))
))
```

6.6 LOGICAL OPERATIONS ON LISTS

A set, in mathematical terms, is an unordered collection of items. Typically, sets contain only numbers which may be mathematically manipulated. Because INTERLISP is a symbolic processing language, sets may contain symbolic information.

A set may be represented as a list. To add an item to a set, we CONS or NCONC it to the list. Deletion of an item is effected by CAR or some other function.

Set theory defines a number of primitive operations on sets: difference, intersection, and union. INTERLISP provides functions that operate on lists as if they were sets. In the following sections, we describe these primitive functions. Then, we look at additional set functions that may be written using them.

6.6.1 Logical Difference

The *logical difference* of two sets, X and Y, is a list consisting of those elements of X that are not members of Y. Consider the following example:

```
←(SETQ states (LIST 'AL 'MD 'NY 'SD 'CA 'IL 'HI 'CT))
(AL MD NY SD CA IL HI CT)

←(SETQ eastern-states (LIST 'AL 'MD 'NY 'CT))
(AL MD NY CT)

←(LDIFFERENCE states eastern-states)
(SD CA IL HI)

←(LDIFFERENCE eastern-states states)
NIL
```

Note that LDIFFERENCE is not a commutative function. **LDIFFERENCE** takes the form

```
Function:       LDIFFERENCE

# Arguments:    2

Arguments:      1) a list, LST1
                2) a list, LST2

Value:          The difference between the two lists
                formed by extracting all elements of the
                first list from the second.
```

We might define LDIFFERENCE as follows:

```
(DEFINEQ
    (ldifference (lst1 lst2)
        (COND
            ((OR
                (NULL lst1)
                (NULL lst2))
                (*
                    If either list is empty, the
                    logical difference with NIL is
                    NIL.
```

```
        NIL)
    ((MEMBER (CAR lst1) lst2)
        (*
            Compare the lists by matching
            down LST1; use recursion.
        )
        (ldifference (CDR lst1) lst2))
    (T
        (*
            The element is not a member of
            LST2, so add it to the list of
            differences.
        )
        (CONS (CAR lst1)
              (ldifference (CDR lst1)
              lst2))))
))
```

Note that LDIFFERENCE returns a new list containing only the members of LST1 which are not found in LST2.

LDIFF: Computing the Difference to a Tail

LDIFF computes the difference between LST1 and LST2. LST2 must be a proper tail of LST1. That is, LST2 is derived by applying CDR some number of times to LST1. Thus, we obtain a list of differences between LST1 and LST2 up to LST2 in LST1. It takes the form

```
Function:       LDIFF
# Arguments:    3
Arguments:      1) a list, LST1
                2) a list which is a tail of LST1, LST2
                3) a result list, LST3
Value:          The differences between LST1 and LST2.
```

If LST3 is not NIL, then the differences between LST1 and LST2 are NCONCed onto LST3. This provides a mechanism for gathering the differences among a number of lists into a single list.

Consider the following example:

```
←(SETQ os '(vms unix vulcan aos multics mvs))
(vms unix vulcan aos multics mvs)
←(SETQ bigos (NLEFT os 1 (LAST os)))
(multics mvs)
```

```
←(LDIFF os bigos)
(vms unix vulcan aos)

←(LDIFF os bigos '(cp/m ms-dos))
(cp/m ms-dos vms unix vulcan aos)

←(LDIFF os '(mcp tenex))
LDIFF: NOT A TAIL
(mcp tenex)

←(LDIFF os)
(vms unix vulcan aos multics mvs)

←(LDIFF os os)
NIL

←(LDIFF os os '(cp/m ms-dos))
(cp/m ms-dos)
```

LDIFF always returns a new list structure unless LST2 is NIL, in which case the value is LST1.

If LST2 is not a tail of LST1, LDIFF generates an error message "LDIFF: NOT A TAIL". LDIFF will terminate on a null check. However, if LST1 is a circular list and LST2 is not a tail, LDIFF goes into an infinite loop.

We might define LDIFF as follows:

```
(DEFINEQ
   (ldiff (lst1 lst2 lst3)
      (COND
         ((EQUAL lst1 lst2)
              (*
                   No differences; return LST3
              )
              lst3)
         ((AND (NULL lst2) (NULL lst3))
              (*
                   LST2 is null; return LST1
              )
              lst1)
         (T
              (PROG (diflst)
                    (*
                        DIFLST is the difference list,
                        which is initialized to LST3, if
                        present.
                    )
                    (SETQ diflst (LAST lst3))
```

```
                (SETQ diflst
                      (CDR
                         (RPLACD diflst
                                 (RPLACD
                                    (CONS (CAR lst1)
                                    diflst)))))
                (COND
                   (lst3 diflst)
                   (T
                     (SETQ diflst
                           (SETQ lst3
                                 (CONS (CAR
                                 lst1))))))
          loop
                (SETQ lst1 (CDR lst1))
                (COND
                    ((EQ lst1 lst2)
                         (*
                              If lists are
                              identically equal,
                              there is no
                              difference, so return
                              LST3.
                         )
                         (RETURN lst3))
                    ((NULL lst1)
                         (*
                              Obviously, LST2 cannot
                              be a tail of the null
                              list. But, when we
                              reach here, we have
                              also exhausted LST1.
                         )
                         (RETURN
                            (ERROR "LDIFF: NOT A
                            TAIL"
                                   lst2))))
                (SETQ diflst
                      (CDR
                         (RPLACD diflst
                                 (RPLACD
                                    (CONS (CAR lst1)
                                    diflst)))))
                (GO loop))))
))
```

6.6.2 Logical Intersection

The *intersection* of two sets, X and Y, is a set consisting of those elements that are members of both X and Y. Consider the following example:

```
←(INTERSECTION states eastern-states)
(AL MD NY CT)

←(INTERSECTION eastern-states states)
(AL MD NY CT)
```

Note that INTERSECTION is a commutative function; that is, the order of the arguments does not affect the result.

INTERSECTION takes the form:

```
Function:       INTERSECTION

# Arguments:    2

Arguments:      1) a list, LST1
                2) a list, LST2

Value:          A list containing elements that appear
                either in LST1 or LST2.
```

We might define INTERSECTION as follows:

```
(DEFINEQ
     (intersection (lst1 lst2)
          (COND
               ((OR
                    (NULL lst1)
                    (NULL lst2))
                    (*
                         No intersection with the null
                         list.
                    )
                    NIL)
               ((MEMBER (CAR lst1) lst2)
                    (*
                         Here we begin to construct the
                         list to be returned. We always
                         choose an element from LST1.
                    )
                    (CONS (CAR lst1)
                          (intersection (CDR lst1)
                          lst2)))
```

```
            (T
                (intersection (CDR lst1) lst2)))
    ))
```

Note that INTERSECTION does not check for duplicates in LST1. Thus, if there are two elements with the same value in LST1, both elements will appear in the result if there is at least one corresponding element in LST2. Sometimes, we want to work with unique sets, e.g., those where there is at most one element of a given value in the set. We can define the function UNIQUE to produce this set as follows:

```
(DEFINEQ
      (unique (x)
          (intersection x x)
      ))
```

Consider the following example:

```
←(SETQ states (LIST 'AL 'NY 'CT 'MI 'AL 'FL 'CT))
(AL NY CT MI AL FL CT)

←(UNIQUE states)
(AL NY CT MI FL)
```

6.6.3 Logical Union

The *union* of two sets, X and Y, is a set consisting of all elements that appear in either X or Y. Consider the example:

```
←(SETQ northern-states (LIST 'MD 'NY 'PA 'RI))
(MD NY PA RI)

←(UNION eastern-states northern-states)
(AL CT MD NY PA RI)

←(UNION northern-states eastern-states)
(PA RI AL MD NY CT)
```

The order of the arguments makes a profound difference in the result. You should consider this carefully if your application depends on the list elements occurring in a particular order.

Note that UNION is not commutative. Its value is a new list consisting of the elements of Y with all elements of X that are not members of Y CONSed to the front. **UNION** takes the form

```
Function:       UNION

# Arguments:    2

Arguments:      1) a list, LST1
                2) a list, LST2

Value:          A new list which is the union of LST1 and
                LST2.
```

We might define UNION as follows:

```
(DEFINEQ
     (union (lst1 lst2)
          (COND
               ((NULL lst1) lst2)
               ((MEMBER (CAR lst1) lst2)
                    (union (CDR lst1) lst2))
               (T
                    (CONS (CAR lst1)
                          (union (CDR lst1) lst2))))
     ))
```

Note that if an element appears twice in LST2, it will also appear twice in the UNION of LST1 and LST2. Sometimes, we want the unique union of two sets. We may define a function to produce the unique union as follows:

```
(DEFINEQ
     (unique-union (x y)
          (intersection (union x y))
     ))
```

6.7 SORTING LISTS

Most lists are constructed without regard to the ordering of the elements. Some applications, however, require a list to be sorted according to some criterion before processing.

Sorting is the process by which a list of items, normally disordered, is placed in order according to some criterion based on the contents of the list. For unordered lists, there is no better procedure than a serial search. If the list does not contain the item sought, we must search the entire list to determine this fact. If the list is ordered, however, certain techniques greatly reduce the searching effort to determine if the item is present. This section will present a number of sorting functions using different algorithms. Further information on sorting may be obtained from Knuth [knut68] and Aho, Hopcroft, and Ullman [aho83].

6.7.1 A Basic Sorting Function

INTERLISP provides the basic function **SORT** to sort lists. It sorts lists by a brute force method of comparing two items in the list at a time. It has the format

```
Function:       SORT
# Arguments:    2
Arguments:      1) a list of items to be sorted, ITEMS
                2) a comparison function, FNCOMPARE
Value:          A sorted list comprised of the items on
                the source list.
```

SORT uses the function specified by FNCOMPARE, which must be a predicate with two arguments, to compare two data items. FNCOMPARE must return T if its first argument belongs before the second, otherwise NIL.

If FNCOMPARE is NIL, SORT uses ALPHORDER (see Section 6.7.3) to lexically order items. Consider the following example:

```
←(SETQ states (LIST 'WA 'MD 'AL 'FL 'CO 'ND 'HI))
(WA MD AL FL CO ND HI)

←(SORT states)
(AL CO FL HI MD ND WA)
```

SORT expects the elements of ITEMS to be atomic values. However, SORT allows you to sort a list whose elements are themselves lists. In this case, the CAR of each list must be atomic. It is passed to ALPHORDER to determine the order of arrangement. To specify this invocation, you must set FNCOMPARE to T. Consider the following example:

```
←(SETQ items '((x 1.0) (b 2.3) (j 3.7) (f 0.4) (q 6.3)))
((x 1.0) (b 2.3) (j 3.7) (f 0.4) (q 6.3))

←(SORT items T)
((b 2.3) (f 0.4) (j 3.7) (q 6.3) (x 1.0))
```

SORT is a destructive function. That is, it modifies the list given as its argument rather than creating a new list. Thus, if the initial ordering of the input list is also important, you should copy the input list before sorting.

We might define SORT as follows:

```
(DEFINEQ
    (sort (items fncompare)
        (PROG (result nxtitems olditems)
            (SETQ olditems items)
```

```
            (SETQ items
                  (CONS (CAR items)
                        (CDR items)))
      loop
            (SETQ nxtitems (CDDR items))
            (SETQ result
                  (CONS
                     (MERGE1 (RPLACD (CDR items))
                             (RPLACD items)
                             fncompare)
                        result))
            (AND
                (SETQ items nxtitems)
                (GO loop))
            (SETQ items result)
      loop2
            (SETQ nxtitems items)
      loop1
            (COND
                (nxtitems
                     (RPLACA nxtitems
                             (MERGE1 (CAR nxtitems)
                                     (CADR nxtitems)
                                     fncompare))
                     (RPLACD nxtitems
                             (CDDR nxtitems))
                     (SETQ nxtitems
                           (CDR nxtitems))
                     (GO loop1)))
            (AND
                (CDR items)
                (GO loop2))
            (RPLACA olditems (CAAR items))
            (RPLACD olditems (CDAR items))
            (RETURN olditems))
))
```

MERGE1 is defined in Section 6.9.

6.7.2 Numeric Sorting

A common application for sorting is to arrange a set of numbers in ascending or descending sequence. We might write a function NUMERIC-SORT as follows

```
(DEFINEQ
    (numeric-sort (items flag)
        (SELECTQ flag
            (ascending (SORT items (FUNCTION
            GREATERP)))
            (descending (SORT items (FUNCTION LESSP)))
            (PROGN
                (PRIN1 "unknown flag")
                (TERPRI)))
    ))
```

6.7.3 Alphameric Sorting

ALPHORDER is a predicate function that is used when sorting lists according to their alphabetical sequence. It takes the form

```
Function:       ALPHORDER
# Arguments:    2
Arguments:      1) an atom, X
                2) an atom, Y
Value:          T, if X occurs before Y; otherwise, NIL.
```

ALPHORDER determines if X occurs before Y. X and Y may be numbers, atoms, or strings. Numbers come before literal atoms and are ordered by magnitude (using GREATERP). Literal atoms and strings are ordered by comparing the character codes of their PRIN1-names. Consider the following examples:

```
←(ALPHORDER 'newton 'benatar)
NIL
←(ALPHORDER 3.141592 3.141593)
T
←(ALPHORDER ''Interlisp-10'' "Interlisp/VAX")
T
```

If neither X nor Y is an atom or a string, ALPHORDER returns T; that is, they are presumed in order because ALPHORDER can make no determination. For example,

```
←(SETQ A1 (ARRAY 3 3))
{ARRAYP}#542224
```

```
←(SETQ A2 (ARRAY 5 5))
{ARRAYP}#542231
←(ALPHORDER A2 A1)
T
```

even though it is clear, by inspection, that A1 occurs physically in memory before A2.

We might define ALPHORDER as follows:

```
(DEFINEQ
   (alphorder (x y)
      (COND
         ((FIXP x)
            (COND
               ((FIXP y)
                  (IGREATERP y x))
               ((FLOATP y)
                  (FGREATERP y x))
               (T T)))
         ((FLOATP x)
            (COND
               ((FIXP y)
                  (FGREATERP (FLOAT y) x)))
               ((FLOATP y)
                  (FGREATREP y x))
               (T T)))
         ((LITATOM x)
            (COND
               ((NUMBERP y) NIL)
               ((LITATOM y)
                  (PROG (index xlen ylen xchar
                  ychar)
                        (SETQ index 0)
                        (SETQ xlen (NCHARS x))
                        (SETQ ylen (NCHARS y))
                  loop
                        (AND
                           (EQ index xlen)
                           (RETURN T))
                        (AND
                           (EQ index ylen)
                           (RETURN NIL))
                        (SETQ index (ADD1 index))
                        (SETQ xchar
                              (NTHCHAR x index))
```

```
                                (SETQ ychar
                                      (NTHCHAR y index))
                                (COND
                                    ((EQ xchar ychar)
                                        (GO loop)))
                                (RETURN
                                    (IGREATERP
                                        (CHCON xchar)
                                        (CHCON ychar)))))
                        (T T)))
                (T T))
        ))
```

The IRM [irm83] notes that ALPHORDER performs no UNPACKs, CHCONs, or NTHCHARs (even though we show it might be coded above using these functions!). Thus, it is several times faster than anything that can be written using these other functions. ALPHORDER is actually implemented in machine language or microcode, but the definition given above (for INTERLISP/370) suggests how it works.

In FranzLisp, this function is known as ALPHALESSP.

6.7.4 Comparing Two Lists

ALPHORDER does not work on lists. Sometimes, we may need to compare two lists to determine their differences. **COMPARELISTS** is a function that compares two lists and prints their differences. Printing is inherent in the operation of COMPARELISTS. It takes the form

```
Function:       COMPARELISTS

# Arguments:    2

Arguments:      1) a list, LST1
                2) a list, LST2

Value:          NIL, but it prints the differences between
                the two lists.
```

Consider the following example:

```
←(COMPARELISTS os bigos)
(vms unix vulcan --)
(multics mvs)
```

Note that COMPARELISTS is subject to the influence of PRINTLEVEL (see Chapter 15).

We might define COMPARELISTS as follows:

```
(DEFINEQ
   (comparelists (lst1 lst2)
      (RESETFORM (PRINTLEVEL 1)
                 (PROG (finish)
                       (COND
                          ((EQUAL lst1 lst2)
                                 (RETURN NIL))
                          ((AND
                                 (NLISTP lst1)
                                 (NLISTP lst2)
                                 (GETD lst1)
                                 (GETD lst2))
                                 (SETQ lst1 (GETD
                                 lst1))
                                 (SETQ lst2 (GETD
                                 lst2))))
                       (COND
                          ((OR
                                 (NLISTP lst1)
                                 (NLISTP lst2))
                                 (PRINT lst1 T)
                                 (PRINT lst2 T)
                                 (GO finish)))
                       (PRIN1 "(" T)
                       (*
                             Print LST1 by comparison
                             with LST2.
                       )
                       (CMPLISTS lst1 lst2)
                       (PRIN1 ")" T)
                       (TERPRI T)
                       (PRIN1 "(" T)
                       (*
                             And vice versa.
                       )
                       (CMPLISTS lst2 lst1)
                       (PRIN1 ")" T)
                       (TERPRI T)
                 finish
                       (RETURN T)))
   ))
```

where CMPLISTS is defined as

```
(DEFINEQ
    (cmplists (xlst ylst)
        (PROG (x y flag dotflag)
        loop1
            (COND
                ((NOT dotflag)
                    (SETQ x (CAR xlst))
                    (SETQ y (CAR ylst)))
                (T
                    (SETQ x xlst)
                    (SETQ y ylst)))
            (COND
                (flag
                    (COND
                        (dotflag (PRIN1 " . " T))
                        (T       (PRIN1 " " T)))))
            (COND
                ((EQUAL x y)
                    (*
                        If the two lists are the
                        same, just print a &.
                    )
                    (PRIN2
                        (COND
                            ((ATOM x) x)
                            (T '&))
                        T))
                ((OR
                    (NLISTP x)
                    (NLISTP y))
                        (*
                            If they are unequal
                            and one is not a list,
                            have PRIN2 display
                            something at the
                            terminal.
                        )
                        (PRIN2 x T))
                (T
                    (*
                        Otherwise,  print "()" and
                        recurse to analyze the
                        sublists.
                    )
                    (PRIN1 "(" T)
```

```
                    (CMPLISTS x y)
                    (PRIN1 ")" T)))
               (SETQ flag T)
               (*
                   FLAG causes CMPLISTS to print ''--''
                   if lists are of different lengths;
                   otherwise, just the CDR.
               )
               (COND
                   (dotflag
                       (RETURN NIL)))
               (COND
                   ((NULL (CDR xlst))
                       (RETURN NIL)))
               (SETQ dotflag (NLISTP (CDR xlst)))
               (COND
                   ((NULL (CDR ylst))
                       (*
                           If YLST expires first,
                           print the tail of XLST.
                       )
                       (COND
                          (dotflag
                             (PRIN1 " . " T)
                             (PRIN2
                                (COND
                                   ((ATOM (CDR xlst))
                                          (CDR xlst))
                                   (T '&))
                                T))
                          ((NULL (CDDR xlst))
                              (SPACES 1 T)
                              (PRIN2
                                 (COND
                                    ((ATOM (CADR xlst))
                                           (CADR xlst))
                                    (T '&))
                                 T))
                          (T
                              (PRIN1 " --" T)))
                       (RETURN NIL)))
                   (SETQ xlst (CDR xlst))
                   (SETQ ylst (CDR ylst))
                   (GO loop1))
))
```

Note that DOTFLAG handles the case where the last element of the list is really a dotted pair.

6.8 LENGTH FUNCTIONS

An essential feature of INTERLISP is that you almost never need to know the length of a list when you are writing your program. There usually comes a time, however, when you must know the length of a list. INTERLISP provides three functions for determining length.

6.8.1 Finding the Length of a List

LENGTH determines the length of its argument. It takes the form

```
Function:       LENGTH
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          The length of the list EXPRESSION as an
                integer.
```

The length is determined by taking successive CDRs until a non-list is found. For example,

```
←(LENGTH '(Cavendish Rutherford Becquerel Curie
Roentgen))
5
←(LENGTH NIL)
 0
```

because NIL is an atom.

```
←(LENGTH 'yankees)
0
```

because atoms are lists of zero length.

```
←(LENGTH '(austin dallas fort-worth))
3
←(LENGTH '(austin dallas fort-worth . galveston))
3
```

because GALVESTON has been CONSed onto the list and so it occupies a CDR cell.

LENGTH is usually hardwired into the virtual machine for efficiency. However, we might define it as follows:

```
(DEFINEQ
     (length (expression)
          (COND
               ((NULL expression) 0)
               ((ATOM expression) 0)
               (T
                    (ADD1 (LENGTH (CDR expression)))))
     ))
```

6.8.2 Counting List Cells

LENGTH works only on the top-level elements of a list. Some algorithms may need to know the total number of cells occupied by a list. These include algorithms that must recursively process sublists of list elements where the time required to execute the algorithm is on the order of the size of the list. Various parsing algorithms fall into this category.

COUNT determines the total number of list cells occupied by its argument. It takes the form

Function:	COUNT
# Arguments:	1
Argument:	1) an S-expression, EXPRESSION
Value:	The number of list cells required to represent EXPRESSION as an integer.

In effect, it applies LENGTH not only to each element of a list but also to each sublist which is an element. COUNT applied to a non-list is 0. Consider these examples:

```
←(COUNT NIL)
0
←(COUNT (CONS 'braves 'brewers))
1
←(COUNT (LIST 'hayes-roth 'lenat 'brown 'feigenbaum))
4
←(COUNT ''Does it believe in strings?")
0
←(COUNT (ARRAY 10))
0
```

and let's look at a complex example:

```
←(SETQ structure
       '(noun-phrase
          ((3 (1000 frame (person sex male)
              startframe)))
          NIL
          (noun-phrase-head NIL
                            (numpers (1 3))) he))
←(LENGTH structure)
5
←(COUNT structure)
22
```

We might define COUNT as follows:

```
(DEFINEQ
    (count (expression)
        (COND
            ((OR
                (NULL expression)
                (ATOM expression)
                (NLISTP expression))
                0)
            (T
                (PLUS
                    (COND
                        ((LISTP (CAR expression))
                            (COUNT (CAR expression)))
                        (T 1))
                    (COUNT (CDR expression)))))
    ))
```

6.8.3 Counting Down a List

When you apply COUNT to a list, you must "touch" every list cell in the list in order to determine the total number. Moreover, COUNT counts every cell in the list. Many times, you merely want to know if a list occupies a minimum number of list cells. **COUNTDOWN** operates like COUNT in that it counts list cells. It takes the form

```
Function:       COUNTDOWN
# Arguments:    2
```

```
Arguments:      1) an S expression, EXPRESSION
                2) a limit, LIMIT

Value:          Either the number of list cells in X or 0.
```

COUNTDOWN counts the list cells that comprise EXPRESSION. It decrements LIMIT as it touches each list cell. COUNTDOWN stops when

1. LIMIT is decremented to 0, whence 0 is returned. This indicates the list has at least LIMIT cells.
2. The end of EXPRESSION is reached while LIMIT is greater than 0. The value returned is equivalent to

```
(IDIFFERENCE limit (COUNT expression))
```

Consider the following examples:

```
←(COUNTDOWN (DOCOLLECT 1 NIL) 100)
0

←(COUNTDOWN (FOR I FROM 1 TO 10 COLLECT (CONS I)) 50)
30
```

where the CLISP statement would yield

```
((1) (2) (3) (4) (5) (6) (7) (8) (9) (10)).

←(COUNTDOWN (CONS 'baltimore 'maryland) 10)
9

←(COUNTDOWN (LIST 'crimson 'red 'magenta 'ochre 'purple) 6)
1
```

Using COUNTDOWN, you may determine how far EXPRESSION has grown toward a predefined limit. This measurement may be used in state space search algorithms where the order of the size of EXPRESSION, which may be a representation of the problem states, is a measure of algorithm efficiency. In this case, we are concerned not only with the LENGTH of EXPRESSION but also the amount of memory that it consumes.

Note also that COUNT will not work with circular lists as it continually cycles through the list. Using COUNTDOWN, you may place a bound on the number of cells to be counted and determine the size of the list relative to that bound.

6.9 MERGING LISTS

In its simplest form, *merging* is the process of taking two ordered lists and creating a single ordered list out of them. Knuth [knut68] and Aho, Hopcroft, and

Ullman [aho83] present additional material on the theory of merging and a number of different algorithms for merging two or more lists. INTERLISP provides two basic functions for merging lists: MERGE and MERGEINSERT.

6.9.1 Merging Two Lists

MERGE destructively combines two sorted lists, X and Y, into a single new list. It takes the format

```
Function:       MERGE

# Arguments:    3

Arguments:      1) a sorted list, X
                2) a sorted list, Y
                3) a comparison function, FNCOMPARE

Value:          A list comprising the destructive merger
                of X and Y.
```

MERGE uses FNCOMPARE to merge the elements of the sorted lists X and Y. FNCOMPARE must be the same function that was used to sort both X and Y. Consider the following examples:

```
←(SETQ x (FOR I FROM 1 TO 10 (COLLECT (RAND 0 1000)))
(844 606 642 606 538 92 883 49 110 865)

←(SETQ y (FOR I FROM 1 TO 10 (COLLECT (RAND 0 1000)))
(533 100 258 677 401 405 104 279 722 926)

←(SORT x (FUNCTION IGREATERP))
(833 865 844 642 606 606 538 110 92 49)

←(SORT y (FUNCTION IGREATERP))
(926 722 677 533 405 401 279 258 104 100)

←(MERGE x y (FUNCTION IGREATERP))
(926 883 865 844 722 677 642 606 606 538 533 405 401 279
 258 110 104 100 92 49)

←(MERGE x y (FUNCTION LESSP))
(883 865 844 642 606 606 538 110 92 49 926 722 677 533
 405 401 279 258 104 100)
```

where we see the lists are not merged because the comparison function is different from the one used to sort the two lists.

```
←(MERGE x y)
(883 865 844 642 606 606 538 110 92 49 926 722 677 533
 405 401 279 258 104 100)
```

because the comparison function is NIL, so alphabetical order is used to compare the elements of the lists.

We might define MERGE as follows:

```
(DEFIN
     (merge (x y fncompare)
          (COND
               ((NULL x) y)
               ((NULL y) x)
               ((NLISTP x)
                    (ERROR "ARG NOT A LIST" x))
               ((NLISTP y)
                    (ERROR "ARG NOT A LIST" y))
               (T
                    (SETQ fncompare
                         (MERGE1 (CONS (CAR x) (CDR x))
                                 (CONS (CAR y) (CDR y))
                                 fncompare))
                    (RPLACA
                         (RPLACD y (CDR fncompare))
                         (CAR fncompare))
                    (RPLACA
                         (RPLACD x (CDR fncompare))
                         (CAR fncompare))))
     ))
```

MERGE1 is defined as follows:

```
(DEFINEQ
     (merge1 (x y fncompare)
          (PROG (result last)
               (OR x (RETURN y))
               (OR y (RETURN x))
               (COND
                    ((SELECTQ fncompare
                         (NIL
                              (ALPHORDER (CAR x) (CAR y)))
                         (T
                              (ALPHORDER (CAAR x) (CAAR y)))
                         (APPLY* fncompare (CAR x) (CAR y)))
                         (SETQ result x)
                         (SETQ x (CDR x)))
                    (T
                         (SETQ result y)
                         (SETQ y (CDR y))))
```

```
                (SETQ last result)
                (COND
                    ((NULL x)
                        (RPLACD last y)
                        (RETURN result)))
                (COND
                    ((NULL y)
                        (RPLACD last x)
                        (RETURN result)))
            loop
                (COND
                    ((SELECTQ fncompare)
                        (NIL
                            (ALPHORDER (CAR x) (CAR y)))
                        (T
                            (ALPHORDER (CAAR x) (CAAR y)))
                        (APPLY* fncompare (CAR x) (CAR y)))
                        (RPLACD last x)
                        (SETQ last x)
                        (SETQ x (CDR x))
                        (COND
                            ((NULL x)
                                (RPLACD last y)
                                (RETURN result))))
                    (T
                        (RPLACD last y)
                        (SETQ last y)
                        (SETQ y (CDR y))
                        (COND
                            ((NULL y)
                                (RPLACD last x)
                                (RETURN result)))))
                (GO loop))
        ))
```

6.9.2 Merging with Insertion

Given a sorted list, X, we often want to insert an item into that list. **MERGEINSERT** inserts a single item into a sorted list. It takes the format

```
Function:       MERGEINSERT
# Arguments:    3
Arguments:      1) a new item, NEW
                2) a sorted list, X
                3) a flag, ONEFLAG
```

```
Value:          A sorted list with the new element
                inserted at the proper location.
```

MERGEINSERT attempts to place the new item, NEW, into the list X in the "best" possible position. Insertion is destructive; that is, X is physically modified. Consider the following example:

```
←(SETQ players '(cobb evers robinson tinker wagner))
(cobb evers robinson tinker wagner)

←(MERGINSERT 'ruth players)
(cobb evers robinson ruth tinker wagner)
```

Usually, we do not want to insert an item into a list if it is already present. If ONEFLAG is T, MERGEINSERT will not modify the list if NEW is already present. For example,

```
←(MERGEINSERT 'evers players T)
(cobb evers robinson ruth tinker wagner)
```

6.10 ASSOCIATION FUNCTIONS

A common use of lists is to relate one value to another value or list of values. Usually, this takes the form (⟨key⟩ ⟨result⟩). KEY is the item to be searched for while RESULT is the value sought. If the result is a pointer, these are sometimes called inverted file indices. INTERLISP calls lists of this form *association lists* because they represent an association between a key and a result. The primary problem with using association lists is searching them.

6.10.1 Searching Lists for Associations

ASSOC searches a list of entries for an item. It returns a list consisting of the item and its associated value. ASSOC takes the form

```
Function:       ASSOC
                SASSOC

# Arguments:    2

Arguments:      1) a search item, KEY
                2) an association list, ALST

Value:          The entry or NIL.
```

The association list is a list of zero or more entries where each entry has the form (usually a dotted pair)

```
(⟨key⟩ . ⟨result⟩)
```

ASSOC searches ALST one entry at a time. It compares the CAR of each entry with KEY. If they are EQ, ASSOC returns the entry. Otherwise, it returns NIL. For example,

```
←(SETQ games
        (LIST (CONS 'bridge 'cards)
              (CONS 'baccarat 'cards)
              (CONS 'chess 'men)
              (CONS 'craps 'dice)
              (CONS 'scrabble 'letters)))
((bridge . cards) (baccarat . cards) (chess . men) (craps .
dice) (scrabble . letters))

←(ASSOC 'chess games)
(chess . men)

←(ASSOC 'backgammon games)
NIL
```

Because ASSOC uses EQ to compare the key against the CAR of each entry, the keys must be atoms.

An alternative form, **SASSOC**, takes the same arguments but uses EQUAL for the comparison. For example,

```
←(SETQ games
        (LIST (LIST (LIST 'bridge) 'cards)
              (LIST (LIST 'baccarat) 'cards)
              (LIST (LIST 'chess) 'men)
              (LIST (LIST 'craps) 'dice)
              (LIST (LIST 'scrabble) 'letters)))
(((bridge) cards) ((baccarat) cards) ((chess) men)
((craps) dice) (scrabble) letters))

←(ASSOC '(bridge) games)
NIL

←(SASSOC '(bridge) games)
((bridge) cards)
```

We might define ASSOC as follows:

```
(DEFINEQ
    (assoc (key alst)
        (PROG NIL
        loop
              (COND
                  ((NLISTP alst)
```

```
                    (COND
                        ((NULL alst)
                            (RETURN NIL))))
                ((EQ (CAAR alst) key)
                    (RETURN (CAR alst))))
            (SETQ alst (CDR alst))
            (GO loop))
    ))
```

6.10.2 Replacing an Association List Value

There are three maintenance operations that we can perform on association lists:

1. Adding a new entry
2. Replacing the value associated with a key
3. Removing an entry

The latter case defaults to setting the value associated with a key to NIL. **PUTASSOC** is used for maintaining association lists. It takes the form

```
Function:       PUTASSOC
# Arguments:    3
Arguments:      1) a search item, KEY
                2) a value, VALUE
                3) an association list, ALST
Value:          The new value.
```

PUTASSOC searches the association list using ASSOC. If it finds an entry with KEY as its CAR, PUTASSOC replaces the associated value with VALUE using RPLACD since the value is the CDR of the entry. Otherwise, it adds a new entry to the association list using (CONS KEY VALUE). If ALST is not a list, PUTASSOC generates an error with the message "ARG NOT LIST". For example,

```
←(PUTASSOC 'checkers 'men games)
men

←games
((bridge . cards) (baccarat . cards) (chess . men) (craps .
dice) (scrabble . letters) (checkers . men))

←(PUTASSOC 'chess 'men-and-board games)
men-and-board
```

```
←games
((bridge . cards) (baccarat . cards) (chess . men-and-
board) (craps . dice) (scrabble . letters) (checkers .
men))
```

We might define PUTASSOC as follows:

```
(DEFINEQ
    (putassoc (key value alst)
        (PROG (entry)
            (COND
                ((NLISTP alst)
                    (ERRORX '(4 T)))
                ((SETQ entry (ASSOC key alst))
                    (RPLACD entry value))
                (T
                    (NCONC alst (CONS key value))
                    (RETURN value))))
    ))
```

6.10.3 Removing an Entry

As we mentioned above, removing an entry from an association list may be considered equivalent to setting the associated value of the key to NIL. Let us create a function DELASSOC that removes an entry from an association list. It takes the form

```
Function:       DELASSOC
# Arguments:    2
Arguments:      1) a search item, KEY
                2) an association list, ALST
Value:          An association list with the entry
                corresponding to KEY removed if it was
                present.
```

We might define a function **DELASSOC** as follows:

```
(DEFINEQ
    (delassoc (key alst)
        (PUTASSOC key NIL alst))
    ))
```

However, this has the unfortunate side effect that, if an entry with KEY as its CAR is not found, then an entry of the form (CONS KEY NIL) will be added to the association list. If we are processing sparse association lists (where the domain of keys is large but the actual key instances are relatively small), the association list will grow with the addition of many entries of the form (⟨key⟩ . NIL). This can have a deleterious effect on program performance since the entire association list must be searched each time. So, let us redefine DELASSOC as follows:

```
(DEFINEQ
    (delassoc (key alst)
        (PROG (entry)
            (COND
                ((NLISTP alst)
                    (ERRORX (LIST 4 'T)))
                ((SETQ entry (ASSOC key alst))
                    (RPLACD entry NIL))
                (T NIL)))
    ))
```

This version is somewhat better in that it adds nothing to the association list if the KEY is not found. However, after some period of time, the list may be "littered" with entries of the form (⟨key⟩ . NIL) as a result of setting previous associations to NIL. What we really intended to do, in many cases, was to remove the entry entirely. The following definition shows how we can do either case:

```
(DEFINEQ
    (delassoc (key alst dflag)
        (PROG (entry)
            (COND
                ((NLISTP alst)
                    (ERRORX (LIST 4 'T)))
                ((SETQ entry (ASSOC key alst))
                    (COND
                        (dflag
                            (RPLACD entry NIL))
                        (T
                            (DREMOVE entry alst))))))
    ))
```

where DFLAG determines whether we set the associated flag to NIL or remove the entry from the association list. If DFLAG is NIL, the entry will be removed.

6.10.4 Adding a Value to an Association List Entry

Often, the value associated with a key in an association list will be a list itself. You may want to add a value to that list. Let us define a function **ADDASSOC** that performs this operation. It takes the form

```
Function:       ADDASSOC
# Arguments:    3
Arguments:      1) a search item, KEY
                2) a value to be added to an entry, VALUE
                3) an association list, ALST
Value:          An association list.
```

ADDASSOC determines if an entry having KEY is present in the association list. If so, it adds VALUE to the value of the entry, converting it to a list if it is not already one. If no entry is present, ADDASSOC creates a new entry consisting of KEY and VALUE.

We can define ADDASSOC which performs this operation:

```
(DEFINEQ
    (addassoc (key value alst)
        (PROG (entry)
            (COND
                ((NLISTP alst)
                    (ERRORX (LIST 4 'T)))
                ((SETQ entry (ASSOC key alst))
                   (RPLACD entry
                           (APPEND (CDR entry)
                                   (COND
                                     ((NLISTP value)
                                        (LIST value))
                                     (T value)))))
                (T NIL)))
    ))
```

6.11 SEARCHING LISTS

We have seen that we can search lists using MEMBER or ASSOC. MEMBER (Section 4.8) compares the key with each element in the list and returns the tail of the list. ASSOC (Section 6.10) compares the key with the CAR of an entry and returns the associated value. MEMBER works on lists whose elements are atoms. ASSOC works on lists having the (⟨key⟩ . ⟨result⟩) format. In the next chapter, we shall see that certain functions also search the property lists of atoms.

LISTGET and LISTPUT operate on the top-level values of atoms which are lists that have a property list format:

```
(key[1] value[1] key[2] value[2] ... key[n] value[n])
```

6.11.1 Searching Lists in Property List Format

LISTGET searches a list two elements at a time. It takes the form

```
Function:       LISTGET
                LISTGET1

# Arguments:    2

Arguments:      1) a list, LST
                2) a search key, KEY

Value:          The next value after the key.
```

LISTGET searches LST using CDDR to access every other element, i.e., it assumes property list format. If an element is EQ to KEY, it returns the next element in the list. We might define it as:

```
(DEFINEQ
     (listget (lst key)
          (COND
               ((NLISTP lst)
                    (ERROR ''ARG NOT LIST'' lst)))
          (PROG (alst entry)
               (SETQ alst (COPY lst))
          loop
               (AND
                    (EQ key (CAR alst))
                    (RETURN (CADR alst)))
               (SETQ alst (CDDR alst)) *
               (AND alst (GO loop))
               (RETURN NIL))
     ))
```

An alternative form, **LISTGET1**, searches the list one CDR at a time. Its definition differs from LISTGET only slightly in that we change the expression indicated by * above to

```
(SETQ alst (CDR alst))
```

6.11.2 Replacing Elements in Place in a List

We may modify a list in place by searching for a value and replacing it. **LISTPUT** acts like LISTGET but replaces the next element if the entry is found. It searches a list using CDDR just like LISTGET. It takes the form

```
Function:       LISTPUT
                LISTPUT1

# Arguments:    3

Arguments:      1) a list, LST
                2) a search key, KEY
                3) a new value, VALUE

Value:          The new value.
```

We might define LISTPUT as follows:

```
(DEFINEQ
     (listput (lst key value)
          (COND
               ((NLISTP lst)
                    (ERROR "not a list")))
          (PROG NIL
          loop
               (AND
                    (EQ key (CAR lst))
                    (RPLACD (CAR lst)
                            (APPEND (LIST value)
                                    (CDDR lst)))
                    (RETURN value))
               (SETQ lst (CDDR lst)) *
               (AND lst (GO loop))
               (RETURN NIL))
     ))
```

An alternative version, **LISTPUT1**, searches the list one CDR at a time. Its definition differs from LISTPUT only slightly in that we change the expression indicated by * above to

```
(SETQ lst (CDR lst))
```

7

Property List Functions

So far, we have seen that atoms can have values. These values may be other atoms, such as numbers, or they may be lists. The value is retrieved by using the literal name of the atom in a function or just giving it to the top-level READ-EVAL-PRINT loop of INTERLISP. The problem with this approach is that an atom standing for some object may really have many characteristics. We want to be able to associate all those characteristics with the atom itself rather than building additional data structures in which we have to look up the name of the atom in order to find the value of the characteristic.

INTERLISP provides us with a convenient method for storing the multiple attributes of an object with the atom which represents that object. This method is called the *property list*. A property list is a conventional list associated with an atom that is composed of pairs of values. Each pair consists of a *property* and a *property value*. Property names and property values are determined by the user. INTERLISP uses some predefined properties for internal functions associated with its utilities.

7.1 CONCEPT OF THE PROPERTY LIST

Suppose we are trying to construct an augmented transition network for natural language parsing. We need to describe each word in our "dictionary" by several attributes. We might require that each word have a class—a part of speech, a number—the person in which that word is used, and a tense—if it is a verb. For example, the word BOY has CLASS = noun and NUMBER = present tense, third person.

We can define the following functions to define a word in the dictionary

```
(DEFINEQ
     (define-word (a-word-definition)
          (MAPCAR (CDR a-word-definition)
```

```
                    (FUNCTION define-word-property))
    ))
(DEFINEQ
    (define-word-property (a-property)
        (PUTPROP (CAR a-word-definition)
                 (CAR a-property)
                 (CDR a-property))
    ))
```

To insert BOY into the dictionary, we execute the following function call

```
(define-word
    '(boy
        (class noun)
        (number (1 3))))
```

This has the effect of placing the property CLASS on BOY's property list with the value NOUN and the property NUMBER with the value (1 3). I determined that the names of the properties would be CLASS and NUMBER. I also determined that the values of the properties would be NOUN and (1 3). Other values for CLASS might be VERB, ADJECTIVE, ADVERB, and so on. Other values for NUMBER might be (2 3), (3 3), and so on.

The property list construct allows us to associate many values with a single atom. With BOY, which represents an object, we can associate many attributes that describe our knowledge about a boy. You are allowed to select the names of the properties and the types of their values. Be careful to distinguish between the value of an atom and the values associated with each of its properties.

Properties would seem to be an efficient way of building a database to describe an object, whether physical or conceptual. Early AI researchers attempted to use the notion of properties as a serious analogy to the attributes of real-world objects. However, they found that representing real-world object characteristics using properties was a difficult task because of the complexity required to model these objects to a sufficient degree of detail. Today, properties are used as a powerful representation mechanism that are used to build complex data structures that capture real-world object characteristics.

7.1.1 The Uniqueness of Atoms

Atoms are unique entities, but they have a scope in relation to how they are used. Consider the following example:

```
←(SETQ color 'black)
black
←(PUTPROP 'color 'number 5)
5
```

```
←(PROG (color)
       (SETQ color 'red)
       (RETURN color))
red

←color
black
```

which shows that the atom COLOR used inside the PROG is a local variable whose existence lasts for the duration of the PROG. However, because it has the same name as an existing atom, INTERLISP saves its value (BLACK) prior to binding a new value inside the PROG expression. The mechanics of saving and restoring values for variables is discussed in Chapter 28.

```
←(PROG (color)
       (PUTPROP 'color 'number 10)
       (RETURN
           (GETPROP 'color 'number)))
10

←(GETPROP 'color 'number)
10
```

Note, however, that changes to a property are not local to a PROG expression or a function definition, even if the atom is a local variable. This is because each atom is a unique object. It may have different values bound to it throughout the execution of the program. The atom COLOR inside the PROG expression is the same atom that was bound at the top level (e.g., external to the PROG). Thus, changes to the property of the atom are global to the entire program because it is exactly the same atom!.

7.2 GETTING A PROPERTY

To retrieve a property from the property list of an atom, you use the function **GETPROP**. GETPROP takes the following form:

```
Function:       GETPROP

# Arguments:    2

Arguments:      1) an atom, ATM
                2) a property name, PROPERTY

Value:          The value of the property if it exists on
                ATM's property list; otherwise NIL.
```

If ATM is not an atom, GETPROP returns NIL. If the property is found on the property list of the atom, GETPROP returns the value associated with that

property name. If the atom does not have a property list or no property by that name exists on the property list, GETPROP returns NIL. Thus, it is not a good idea to choose NIL as the default value of a property because it is difficult to distinguish whether the property has been initialized or really has the value NIL.

Suppose I want to retrieve the property CLASS of the word BOY from our dictionary. I would say

```
←(GETPROP 'boy 'class)
noun
```

Note that GETPROP returns the value associated with property which, in this case, is the atom NOUN.

If I wished to retrieve the NUMBER of BOY, I would say

```
←(GETPROP 'boy 'number)
(1 3)
```

Note that GETPROP returns the exact value of the property. In one case, it was an atom while in the other it was a list.

If I asked for the TENSE of BOY:

```
←(GETPROP 'boy 'tense)
NIL
```

because the word BOY does not have a tense since it is not a verb.

Often, we will initialize the values of properties on a property list to NIL because we will set them later on in the course of the program. In this case, GETPROP will return the value NIL when we access that property. To distinguish whether or not the property exists versus whether it exists with value NIL, we must use the following form [irm78]:

```
(MEMBER property (GETPROPLIST atom))
```

We might define GETPROP as follows:

```
(DEFINEQ
    (getprop (atm property)
        (COND
            ((LITATOM atm)
                (PROG (proplst)
                    (SETQ proplst (GETPROPLIST
                    atm))
                loop
                    (COND
```

```
                    ((OR
                         (NLISTP proplst)
                         (NLISTP (CDR
                         proplst)))
                         (RETURN NIL))
                    ((EQUAL (CAR proplst)
                            property)
                         (*
                              Extract value of
                              property from the
                              list.
                         )
                         (RETURN (CADR
                         proplst))))
               (SETQ proplst (CDDR proplst))
               (GO loop)))
          (T NIL))
    ))
```

7.2.1 Getting the Entire Property List

GETPROP is used for retrieving the value of a single property. Sometimes, I want to manipulate the entire property list of an atom. I can retrieve the entire property list by executing **GETPROPLIST**. GETPROPLIST takes the form

Function:	GETPROPLIST
# Arguments:	1
Argument:	1) an atom, ATM
Value:	A list, possibly NIL, representing the entire property list associated with ATM.

If the argument is not an atom, GETPROPLIST generates an error: ARG NOT LITATOM. For example,

```
←(SETQ word 'boy)
boy
←(GETPROPLIST word)
((class noun) (number (1 3)))
```

but

```
←(SETQ word '(boy))
(boy)
```

```
←(GETPROPLIST word)
ARG NOT LITATOM
```

Here is a more complex example:

```
←(PRINTDEF (GETPROPLIST 'DECLARE))
(10MACRO (X (PROGN
                (MAPC X
                      (FUNCTION
                          (LAMBDA (X MACROX)
                              (COND
                                  ((NEQ (CAR X) 'CLISP:)
                                      (EVAL X))))))
                (REFRAME)
                'INSTRUCTIONS))
CLISPWORD (FORWARD . declare))
```

which has two properties: 10MACRO and CLISPWORD. The value of 10MACRO is the macro definition for the atom DECLARE.

7.3 PUTTING PROPERTIES

The corresponding functions for putting properties are PUTPROP and SETPROPLIST. **PUTPROP** takes the following form:

```
Function:       PUTPROP
# Arguments:    3
Arguments:      1) an atom, ATM
                2) a property name, PROPERTY
                3) a value, VALUE
Value:          The value of the property.
```

PUTPROP puts the specified property on the property list of the atom with the given value. If the property already exists, the given value replaces the old value. If the first argument is not an atom, PUTPROP generates an error message, ARG NOT LITATOM. For example,

```
←(PUTPROP 'maryland 'capitol 'annapolis)
annapolis
←(GETPROPLIST 'maryland)
(capitol annapolis)
```

because the property CAPITOL did not previously exist. Now, by applying PUTPROP again

```
←(PUTPROP 'maryland 'capitol 'baltimore)
(new CAPITOL property for MARYLAND)
baltimore
```

because we performed this operation at the top level of the READ-EVAL-PRINT loop.

```
←(GETPROPLIST 'maryland)
(capitol baltimore)
```

A Definition for PUTPROP

We might define PUTPROP as follows:

```
(DEFINEQ
     (putprop (atm property value)
          (COND
               ((NULL atm)
                    (*
                         If ATM is null, generate an
                         error.
                    )
                    (ERRORX (LIST 7 (LIST atm
                    property))))
               ((NOT (LITATOM atm))
                    (ERROR "ARG NOT LITATOM" atm)))
          (PROG (proplst)
                (SETQ proplst (GETPROPLIST atm))
          loop
                (COND
                   ((NLISTP proplst xprop atemp)
                      (COND
                         ((AND
                             (NULL proplst)
                             xtemp)
                             (*
                                  We have encountered
                                  the end of the
                                  property list where
                                  there are an even
                                  number of entries,
                                  i.e., one value for
                                  every property name.
                             )
                             (SETQ atemp
                                   (LIST property
                                   value))
```

```
                                (RPLACD (CDR xprop) atemp))
                                (RETURN value)))
                                (*
                                     Note: the property
                                     list was initially NIL
                                     or a non-list, so we
                                     add the new property
                                     at the beginning.
                                ))
                    ((NLISTP (CDR proplst))
                         (*
                                The property list
                                terminates at an odd
                                position, e.g., a name
                                followed by no value or a
                                name with the value CONSed
                                to it. So add it at the
                                beginning.
                                ))
                    ((EQ (CAR xprop) property)
                         (*
                                The property name is found
                                in the property list, so
                                just replace the value.
                                )
                                (RPLACA (CDR xprop) value)
                                (RETURN value))
                    (T
                          (SETQ xprop proplst)
                          (SETQ proplst (CDDR xprop))
                          (GO loop)))
               (*
                   Add new property and value to front
                   of property list.
               )
               (SETQ atemp
                     (CONS property
                           (CONS value
                                 (GETPROPLIST atm))))
               (SETPROPLIST atm atemp)
               (RETURN value))
))
```

7.3.1 Assigning Multiple Properties

In many cases, you may wish to assign multiple properties to an atom. This can be a tedious operation if each assignment requires a single invocation of PUTPROP. INTERLISP provides an alternative, **PUTPROPS**, to perform multiple property assignments. It takes the form

```
Function:       PUTPROPS

# Arguments:    3-N

Arguments:      1) an atom, ATM
                2) a property name, PROPERTY
                3) a value, VALUE
                4-N) property name/value pairs

Value:          NIL
```

PUTPROPS is an NLAMBDA, nospread function.

PUTPROPS takes successive pairs of property names and values and places them on the property list of the specified atom. Its value is NIL. Consider the following example:

```
←(PUTPROPS maryland
           governor harry-hughes
           (senators (charles-mathias paul-sarbanes)
           counties (kent harford))
NIL
←(GETPROPLIST 'maryland)
(capitol annapolis governor harry-hughes senators
           (charles-mathias paul-sarbanes) counties
           (kent harford))
```

Note that the arguments to PUTPROPS are not evaluated. Thus, you do not need to quote MARYLAND and any of the arguments.

7.3.2 Setting the Property List

You may also construct the property list yourself and then assign it to an atom via **SETPROPLIST**. SETPROPLIST takes the form

```
Function:       SETPROPLIST

# Arguments:    2

Arguments:      1) an atom, ATM
                2) an S-expression, EXPRESSION
```

```
Value:          The property list which is the value of
                EXPRESSION.
```

SETPROPLIST sets the property list of ATM to be the value of its second argument. If the first argument is not an atom, SETPROPLIST displays the error message ARG NOT LITATOM. If the first argument is NIL, SETPROPLIST displays the error message ATTEMPT TO RPLAC NIL. However, if the S-expression is also NIL, SETPROPLIST merely returns NIL. For example,

```
←(SETPROPLIST 'maryland
              (LIST 'governor 'hughes
                    'senators (LIST 'mathias
                    'sarbanes)))
(governor hughes senators (mathias sarbanes))
←(GETPROPLIST 'maryland)
(governor hughes senators (mathias sarbanes))
```

Note that any property list that previously existed for MARYLAND is completely replaced by SETPROPLIST.

7.3.3 Defining a Property for Multiple Atoms

You may wish to assign the same property to many atoms with the same or different values. **DEFLIST** takes the form

```
Function:       DEFLIST
# Arguments:    2
Arguments:      1) a list of atom-value pairs, LST
                2) a property name, PROPERTY
Value:          NIL
```

LST is a list of elements each of which has the form

```
(⟨atom⟩ . ⟨value⟩)
```

DEFLIST puts PROPERTY on the property list of every atom which is the CAR of an entry on LST with the value which is the CDR of that entry. Consider the following example:

```
←(DEFLIST '((maryland hughes)
            (virginia robb)
            (massachusetts dukakis)
            (texas white))
          'governor)
```

```
NIL

←(GETPROPLIST 'maryland)
(capitol annapolis governor hughes)

←(GETPROPLIST 'texas)
(governor white)
```

A Definition for DEFLIST
We might define DEFLIST as follows:

```
(DEFINEQ
     (deflist (lst property)
          (PROG NIL
          loop
                (COND
                   ((NLISTP lst)
                         (RETURN NIL)))
                (PUTPROP (CAAR lst)
                         property
                         (CADAR lst))
                (SETQ lst (CDR lst))
                (GO loop))
     ))
```

7.4 MODIFYING PROPERTY LISTS

INTERLISP provides three functions for modifying property lists:

ADDPROP adds a new property
REMPROP removes a property
CHANGEPROP changes the name, but not the value, of a property

7.4.1 Adding a Property

A property may have a list (or other data structure) as its value. Adding a new value to that list usually requires the creation of a new list structure with the value added to the front or back of the list. For example,

```
←(SETQ property-value (GETPROP atm prop))

←(PUTPROP atm prop (CONS property-value newvalue))
```

or

```
(PUTPROP atm prop (APPEND property-value newvalue))
```

To avoid creating a new list structure, INTERLISP provides **ADDPROP** to add a value to a list which is the value of the property. It takes the form

```
Function:       ADDPROP
# Arguments:    4
Arguments:      1) an atom, ATM
                2) a property name, PROPERTY
                3) a new value, NEW
                4) a flag, FLAG
Value:          The value of the new property.
```

FLAG controls the terminus at which the property will be added. If FLAG is T, the new property is CONSed to the front of the list; otherwise, it is NCONCed to the end of it. If no property exists for the atom, the effect is the same as

```
(PUTPROP atm prop (LIST new))
```

For example,

```
←(ADDPROP 'maryland 'representatives 'byron)
(byron)
←(GETPROP 'maryland 'representatives)
(byron)
←(ADDPROP 'maryland 'representatives 'barnes T)
(barnes byron)
←(GETPROP 'maryland 'representatives)
(barnes byron)
←(ADDPROP 'maryland 'representatives 'hoyer)
(barnes byron hoyer)
←(GETPROP 'maryland 'representatives)
(barnes byron hoyer)
```

A Definition for ADDPROP

We might define ADDPROP as follows:

```
(DEFINEQ
    (addprop (atm property value flag)
        (COND
            ((NULL atm)
                (ERROR "ATTEMPT TO RPLAC NIL"))
            ((NOT (LITATOM atm))
```

```
          (ERROR "ARG NOT LITATOM" atm)))
(PROG (x proplst)
      (SETQ x atm)
loop
      (COND
         ((NLISTP (CDR x))
            (*
               Note that the CDR cell of an
               atom contains the pointer to the
               property list form that atom.

               Here, we have reached the end of
               the property list, so we just
               RPLACD a new entry onto its end.
            )
            (SETQ proplst
                  (LIST property
                        (SETQ value
                              (LIST value))))
            (RPLACD x proplst))
         ((EQ (CADR x) property)
            (*
               Found the property name in the
               property list.
            )
            (RPLACA (CDDR x)
                    (SETQ value
                          (COND
                             (flag
                                (CONS value
                                      (CADDR x)))
                             (T
                                (NCONC1 (CADDR x)
                                        value))))
                    ))
         ((SETQ x (CDDR x))
               (*
                    Have not found property;
                    advance along property
                    list.
               )
               (GO loop))
         (T
               (*
                    Property list ended on a
```

```
                                property name with no
                                value.
                            )
                            (SETQ proplst
                                  (CONS property
                                        (CONS
                                           (SETQ value
                                                 (LIST value))
                                           (CDR atm))))
                            (RPLACD atm proplst)))
                    (RETURN value))
        ))
```

7.4.2 Removing a Property

You may remove a property from the property list via **REMPROP**. It takes the form

```
Function:       REMPROP
# Arguments:    2
Arguments:      1) an atom, ATM
                2) a property name, PROPERTY
Value:          The property name or NIL.
```

REMPROP removes all occurrences of the specified property and its value from the atom's property list. If any occurrences are found, its value is the name of the property. Otherwise, it returns NIL. Note that multiple occurrences of a property may appear in a property list if the list was assigned to the atom via SETPROPLIST. Consider the following examples:

```
←(GETPROPLIST 'maryland)
(capitol   annapolis   governor hughes senators
(mathias      sarbanes) counties (kent harford))

←(REMPROP 'maryland 'governor)
governor

←(GETPROPLIST 'maryland)
(capitol annapolis senators (mathias sarbanes) counties
(kent harford))

←(REMPROP 'maryland 'population)
NIL
```

A Definition for REMPROP
We might define REMPROP as follows:

```
(DEFINEQ
  (remprop (atm property)
    (COND
         ((NULL (LITATOM atm))
               (ERROR "ARG NOT LITATOM" atm)))
    (PROG (proplst newproplst value)
          (SETQ proplst
               (GETPROPLIST atm))
    loop
          (COND
            ((OR
                (NLISTP proplst)
                (NLISTP (CDR proplst)))
                (RETURN value))
            ((EQUAL (CAR proplst) property)
                   (SETQ value property)
                   (COND
                     (newproplst
                       (RPLACD (CDR newproplst)
                               (CCDR proplst)))
                     (T
                       (SETPROPLIST atm
                                    (CDDR newproplst)))
                   ))
            (T
              (SETQ proplst
                   (CDDR (SETQ newproplst proplst)))))
          (GO loop))
    ))
```

7.4.3 Removing the Property List

To remove multiple properties, INTERLISP provides the function **REMPROPLIST**. It takes the form

```
Function:       REMPROPLIST

# Arguments:    2

Arguments:      1) an atom, ATM
                2) a list of property names, PROPLST

Value:          NIL
```

REMPROPLIST removes all the properties whose names appear in PROPLST from the property list of the specified atom.

To remove the entire property list of an atom, you may use one of the following function calls:

```
(SETPROPLIST ⟨atom⟩ NIL)
(REMPROPLIST ⟨atom⟩ (PROPNAMES ⟨atom⟩))
```

Note: Many atoms in INTERLISP already have property lists with properties used by various system packages. You should be careful not to delete such properties as they may cause the system to malfunction (or cease to function). You may use SYSPROPS to obtain a list of all the system property names.

Heeding this advice, we should rewrite the second expression above as follows

```
(MAPC (PROPNAMES ⟨atom⟩)
      '(LAMBDA (name)
               (COND
                    ((MEMBER name (SYSPROPS)) NIL)
                    (T (REMPROP ⟨atom⟩ name)))))
```

Consider the following example where we force the property list to have multiple copies of a property name

```
←(SETPROPLIST  x '(a b c d a b e f g h a b))
(a b c d a b e f g h a b)
```

The property name A is repeated three times in this property list. Now, we may remove A via REMPROPLIST

```
←(REMPROPLIST 'x '(A))
NIL
←(GETPROPLIST 'x)
(c d e f g h)
```

7.4.4 Changing Property Names

The foregoing functions changed the structure of the property list by adding or deleting a property and its associated value. Sometimes, you want to keep the value but change the name of the property. **CHANGEPROP** takes the form

```
Function:       CHANGEPROP
# Arguments:    3
```

```
Arguments:      1) an atom, ATM
                2) a property name, PROPERTY1
                3) a property name, PROPERTY2
Value:          The atom name.
```

CHANGEPROP changes the name of property PROPERTY1 to the name given by PROPERTY2 on the property list of ATM. If PROPERTY1 does not exist on ATM's property list, it returns NIL. If ATM is not a literal atom, CHANGEPROP generates an error with the message "ARG NOT LITATOM".

Consider the following example:

```
←(GETPROPLIST 'maryland)
(capitol annapolis governor hughes senators (mathias
sarbanes))
```

Note that we have used the property name SENATORS to represent the United States senators. However, Maryland has a bicameral legislature in which the upper house is also called the Senate. So, let us rename the property SENATORS to US-SENATORS to avoid confusion.

```
←(CHANGEPROP 'maryland 'senators 'us-senators)
maryland
←(GETPROPLIST 'maryland)
(capitol annapolis governor hughes us-senators (mathias
sarbanes))
```

A Definition for CHANGEPROP

We might define CHANGEPROP as follows:

```
(DEFINEQ
     (changeprop (atm property1 property2)
          (COND
               ((NOT (LITATOM atm))
                    (ERRORX (LIST "ARG NOT LITATOM"
                    atm)))
               ((SETQ property1
                      (MEMBER property1 (GETPROPLIST
                      atm)))
                      (*
                         Only change the name if
                         property1 is actually present in
                         the property list.
                      )
```

```
                    (RPLACA property1 property2)
                    atm))
    ))
```

7.5 OBTAINING THE PROPERTY NAMES OF AN ATOM

PROPNAMES allows you to obtain a list of the property names associated with an atom. It takes the form

```
Function:       PROPNAMES
# Arguments:    1
Arguments:      1) an atom, ATM
Value:          A list of the property names.
```

PROPNAMES is useful when you want to apply a function to all the properties of an atom. For example, to delete all of the properties associated with an atom, we can use the following expression:

```
(REMPROPLIST atm (PROPNAMES atm))
```

We might print the properties and their associated values using the following expression

```
(MAPC (PROPNAMES atm)
      '(LAMBDA   (name)
          (PRIN1   name)
          (SPACES   2)
          (PRINTDEF (GETPROP atm name))))
```

A Definition for PROPNAMES

We might define PROPNAMES as follows:

```
(DEFINEQ
    (propnames (atm)
        (PROG (propnamelst proplst)
              (SETQ proplst (GETPROPLIST atm))
              (SETQ propnamelst NIL)
        loop
              (COND
                  ((NLISTP proplst)
                      (RETURN propnamelst)))
              (SETQ propnamelst
                  (APPEND propnamelst (CAR proplst)))
```

```
                (SETQ proplst (CDDR proplst))
                (GO loop))
    ))
```

7.5.1 Obtaining the System Property Names

INTERLISP has many predefined property names that are used by various system packages. You may obtain a list of the system property names by executing **SYSPROPS** which takes the form

```
Function:       SYSPROPS
# Arguments:    0
Arguments:      NIL
Value:          A list of system property names.
```

For example, the following list was obtained by executing SYSPROPS under the Fugue release on a Xerox 1100 Scientific Information Processor:

```
←(SYSPROPS)
(BYTEMACRO ALTOMACRO JMACRO VAXMACRO DMACRO 10MACRO
VARTYPE HASDEF FILEPKGCONTENTS PROPTYPE ALISTTYPE DELDEF
EDITDEF PUTDEF GETDEF WHENCHANGED NOTICEFN NEWCOMFN
PRETTYTYPE DELFROMPRETTYCOM ADDTOPRETTYCOM ACCESSFN ACS
ADVICE ADVISED ALIAS AMAC ARGNAMES BLKLIBRARYDEF BRKINFO
BROADSCOPE BROKEN BROKEN-IN CLISPCLASS CLISPCLASSDEF
CLISPFORM CLISPIFYISPROP CLISPINFIX CLISPISFORM
CLISPISPROP CLISPNEG CLISPTYPE CLISPWORD CLMAPS CODE
CONVERT COREVAL CROPS CTYPE EDIT-SAVE EXPR FILE
FILECHANGES FILEDATES FILEDEF FILEGROUP FILEHISTORY
FILEMAP FILETYPE GLOBALVAR HISTORY I.S.OPR I.S.TYPE INFO
LASTVALUE LISPFN MACRO MAKE NAMESCHANGED NARGS OLDVALUE
OPD READVICE SETFN SUBR UBOX UNARYOP VALUE DEF
CLISPBRACKET)
```

7.6 EXTRACTING A PROPERTY SUBLIST

You may extract a sublist of the property list using **GETLIS**, which takes the form

```
Function:       GETLIS
# Arguments:    2
```

```
Arguments:      1) an atom, ATM
                2) a list of properties, PROPLST

Value:          The tail of the property list for ATM.
```

GETLIS searches the property list of ATM using the values found on PROPLST which may be an atom or a list of properties. It returns the tail of the property list beginning with PROPLST. If PROPLST is an atom, then that property name is the CAR of the result. If PROPLST is a list, GETLIS uses each element on PROPLST as a search key until one matches a property of ATM. The tail of the property list beginning with that element is returned.

If no element of PROPLST is found on the property list of ATM, GETLIS returns NIL. For example,

```
←(GETLIS 'maryland '(us-senators))
(us-senators (mathias sarbanes) counties (kent harford))

←(GETLIS 'maryland 'governor)
NIL

←(GETLIS 'maryland 'capitol)
NIL
```

Note that if PROPLST is an atom, as in the case above, GETLIS returns NIL. Thus, you cannot determine whether or not the property exists.

A Definition for GETLIS

We might define GETLIS as follows:

```
(DEFINEQ
    (getlis (atm proplst)
        (PROG (atmproplst)
            (SETQ atmproplst (GETPROPLIST atm))
        loop
            (COND
                ((OR
                    (NLISTP atmproplst)
                    (MEMBER (CAR atmproplst)
                    proplst))
                    (RETURN atmproplst)))
            (SETQ atmproplst (CDR atmproplst))
            (GO loop))
    ))
```

Note that GETLIS makes no distinction between the property names and their values when it searches the atom's property list.

8

Function Definition and Evaluation

INTERLISP embodies the mathematical definition of a function as a procedure or specification for action. A function takes zero or more arguments as inputs and produces one or more values as outputs. Unlike mathematical functions, however, INTERLISP functions exist within a program—a collection of functions gathered to accomplish a purpose—and, so, may have side effects on global variables defined within the program.

In INTERLISP, there are three ways to create a function for evaluation:

1. Use DEFINEQ or PUTD to attach the definition to an atomic symbol (Section 8.2.2)
2. Place the definition on an atom's property list under one of the properties EXPR, FEXPR, or MACRO (Section 7.3)
3. Dynamically bind a definition to an atom through the FUNARG mechanism (Section 12.4)

Because atoms have both value cells and function definition cells, we cannot determine whether an arbitrary data object is to be used as a variable or a function. Usage is determined by how the data object is evaluated (e.g., which primitive functions are applied to it) because the value cell and the function definition cell may both have non-NIL values.

The primary focus of this chapter is to discuss the definition and evaluation of user-defined functions. Several system functions allow you to obtain information about a function and its arguments. Most functions do not retain any historical information about previous invocations unless you explicitly build their structures to do so. Generators are a simple mechanism for maintaining status information across function invocations.

8.1 FUNCTION TYPES

INTERLISP provides two types of functions: EXPRs and SUBRs.

An **EXPR** is a function that is written in INTERLISP; it is *interpreted*. Each literal atom has a function definition cell in which a function definition may be stored. A function definition is a pointer to a list that describes the procedure that the function performs. Note that the atom may also have a defined value that is independent of its function definition.

A **SUBR** is a function that is *hand-coded* in machine language (INTERLISP/370) or is provided by the underlying virtual machine firmware (INTERLISP-D). Compiled functions (see below) are treated as SUBRs.

The distinction is important because the functions PUTD and GETD operate differently upon the two types of functions. You cannot define a SUBR with either DEFINE or DEFINEQ. When you compile an EXPR function, you effectively convert it to a SUBR when it is loaded into memory.

Applying GETD to an EXPR function name returns the list comprising the function definition. Applying GETD to a SUBR returns a dotted pair that has the following characteristics:

The CAR of the dotted pair is an encoded form of

1. the argument type
2. the number of arguments

The CDR of the dotted pair is the address of the first instruction of the function.

INTERLISP allows us to modify the basic concept of a function in two respects: whether or not the arguments are evaluated, and whether or not the function has a definite number of arguments.

8.1.1 To Evaluate or Not

Normally, arguments passed to functions are evaluated to yield their respective values which are then used by the function. This is called a LAMBDA-type function. Consider the following example:

```
←(SETQ integer-1 10)
10
←(SETQ integer-2 20)
20
←(IDIFFERENCE integer-1 integer-2)
-10
```

IDIFFERENCE is a *LAMBDA-type* function. The values that IDIFFERENCE receives are the numbers 10 and 20 which are the values of the arguments when they are evaluated. This mechanism is known as *call by value* in other languages.

You may specify that a function's arguments not be evaluated when the function is invoked. This is known as an *NLAMBDA-type* function and the mechanism is similar to a *call by reference* in other languages. DEFINEQ (see below) is an NLAMBDA-type function. Consider the following example:

```
(DEFINEQ
     (exchange (pair)
          (LIST (CADR pair) (CAR pair))
     ))
```

The argument to DEFINEQ is not evaluated but is passed to DEFINEQ for internal evaluation. Thus, the value passed to DEFINEQ is the list

```
(exchange (pair) (LIST (CADR pair) (CAR pair)))
```

When FNTYP (see Section 8.1.5) is applied to NLAMBDA-type functions, it returns the type of the function prefixed by the letter F: FEXPR or FSUBR.

8.1.2 To Spread or Not

Normally, a function is defined with a definite number of arguments that are enumerated in its parameter list. INTERLISP matches the arguments specified in the function invocation with the parameters specified by the function definition (i.e., we say the arguments are *spread* across the parameters). If there are not enough arguments to satisfy all the parameters, INTERLISP substitutes NIL for the remaining parameters. If there are more arguments than parameters, the excess arguments are ignored (but there are a few exceptions, e.g., see QUOTE).

Sometimes, we want to pass an indefinite number of arguments to a function. The number and type of arguments may be dependent on the arguments themselves. How the arguments are to be evaluated is determined within the function itself. This is called a *nospread* function.

To define a nospread function, we give a single variable, not enclosed in parentheses, after the LAMBDA or NLAMBDA declaration. For example,

```
(DEFINEQ
     (add
          (LAMBDA x
               (PROG (sum lst)
                     (SETQ sum 0)
                     (SETQ lst (copy x))
                     (AND (NULL lst)(RETURN sum))
               loop
                     (SETQ sum (plus sum (car x)))
```

```
                    (AND
                        (SETQ lst (CDR lst))
                        (GO loop))
                    (RETURN sum))
))))
```

where X is a list of all the arguments presented in the calling sequence to the function. The arguments are evaluated or not depending on whether the function is defined to be LAMBDA or NLAMBDA. For example,

```
←(ADD 10 20 30)
```

would set X to the list (10 20 30).

When FNTYP is given the name of a function which does not spread its arguments, it returns the type of the function with an asterisk (*) appended to the type name.

8.1.3 Compiled Functions

Most INTERLISP functions are interpreted. Even in the best of circumstances, interpretation can be a lengthy process. Once a function is debugged and ready for production use, you may compile the function. INTERLISP provides a compiler (see Chapter 31) that converts the INTERLISP source code into the underlying computer's machine language. When FNTYP is given the name of a function that has been compiled, it returns the type of the function preceded by the letter C: CEXPR or CFEXPR. For the purposes of applying GETD to compiled functions, they are treated like SUBRs.

8.1.4 Summary of Function Types

In summary, there are 12 types of functions supported by INTERLISP. These are shown in the following table:

EXPR	**SUBR**
EXPR	SUBR
EXPR*	SUBR*
FEXPR	FSUBR
FEXPR*	FSUBR*
CEXPR	
CEXPR*	
CFEXPR	
CFEXPR*	

Note that SUBRs, because they are written in machine language, already exist in a compiled state.

8.1.5 Determining the Function Type

You may determine the type of a function by executing FNTYP. **FNTYP** takes the following form

```
Function:       FNTYP
# Arguments:    1
Argument:       1) a function name, FN
Value:          The function type.
```

The argument of FNTYP may be either the name of a defined function or a function definition itself. The value of FNTYP is an atom taken from the table below. Otherwise, it returns NIL. For example,

```
←(FNTYP 'GETD)
CEXPR
←(FNTYP 'DEFINE)
CEXPR
←(FNTYP 'DEFINEQ)
CFEXPR*
←(FNTYP '(LAMBDA (pair)
                 (LIST (CADR pair) (CAR pair))))
EXPR
```

where FNTYP has checked the function definition (indicated by the LAMBDA) to determine the function type.

```
←(SETQ a-function 'GETD)
←(FNTYP a-function)
CEXPR
```

FNTYP may also return the atom FUNARG if the argument is a function argument expression (see Section 12.4).

The value of FNTYP is one of the following twelve literal atoms, based on the type of the function:

	Expressions	*Compiled*	*Built-In*
Lambda-Spread	EXPR	CEXPR	SUBR
Nlambda-Spread	FEXPR	CFEXPR	FSUBR
Lambda-Nospread	EXPR*	CEXPR*	SUBR*
Nlambda-Nospread	FEXPR*	CFEXPR*	FSUBR*

The types in the Built-In column are returned only for INTERLISP-10 and INTERLISP/370.

A Definition for FNTYP

We might define FNTYP as follows (after INTERLISP/370):

```
(DEFINEQ
    (fntyp (fn)
        (SELECTQ (NTYP (MKFNIND fn))
            ((0 4) 'FSUBR*)
            ((1 5) 'FSUBR)
            ((2 6) 'SUBR*)
            ((3 7) 'SUBR)
            (8     'FEXPR*)
            (9     'FEXPR)
            (10    'EXPR*)
            (11    'EXPR)
            (12    'CFEXPR*)
            (13    'CFEXPR)
            (14    'CEXPR*)
            (15    'CEXPR)
            (20    'FUNARG)
            NIL)
    ))
```

where the number is an indication of the datatype for the atom.

8.2 DEFINING FUNCTIONS

You may define a new function in INTERLISP using either DEFINE or DEFINEQ. Let us discuss DEFINEQ first and, then, explore the generalized version represented by DEFINE.

8.2.1 Syntax of a Function Definition

A function definition consists of the following components:

The name of the function
A declaration of function type: LAMBDA or NLAMBDA
A parameter list or NIL
The body of the function

These items are combined together in a list which is passed to either DEFINE or DEFINEQ.

The name of the function must be a symbolic atom that conforms to the naming conventions of the particular INTERLISP system that you are using.

LAMBDA or NLAMBDA determines whether the function's arguments are evaluated or not when the function is invoked.

The parameter list is a list of the names of symbolic atoms, usually enclosed in parentheses, that represent the arguments to the function. Each symbolic atom may appear in the body of the function in which case we say that the atom is *local* to the function. The parameter list may be represented by the atom NIL which means that the function expects no arguments. The parameter list may consist of a single atom which indicates that the arguments are not spread. In this case, when the function is invoked, all arguments specified in the argument list are gathered into a list which is bound to the single atom.

The body of the function is a sequence of INTERLISP statements that specifies the procedure of the function. Variables appearing in the function body that do not appear in the parameter list are assumed to be global variables for this function, although they may be local and bound in a function which invoked this function. When the function is executed, if the value of these variables is changed, we say that the function has a *side effect*. That is, the value of the variable persists beyond the execution of the function because the variable is defined outside the function. The value a function gives back when it is executed is called the *value returned*. This value is the result of the last statement executed within the function body, which may not necessarily be the last physical expression in the function body.

8.2.2 Defining a Function: DEFINEQ

DEFINEQ is used to define one or more functions. It takes the form

```
Function:       DEFINEQ
# Arguments:    1-N
Arguments:      1) a list of one or more function forms,
                   FNS
Value:          The names of the function that are
                defined.
```

DEFINEQ is an NLAMBDA, nospread function. It takes an indefinite number of arguments that are not evaluated. Each entry in FNS must take the form required by DEFINE (see below). If it does not, DEFINEQ (or DEFINE) issues an error message "INCORRECT DEFINING FORM". The File Package writes functions to symbolic files using DEFINEQ so that, when read in via LOAD, they will be defined anew in your virtual memory.

DEFINEQ calls DEFINE with individual entries from FNS.

8.2.3 Defining a Function: DEFINE

DEFINE is the LAMBDA-spread version for defining a function. It takes the form

```
Function:       DEFINE
# Arguments:    1
Arguments:      1) a list of lists, X
Value:          The names of functions defined as
                described in X.
```

Each element of the list X is itself a list which takes one of two forms:

1. (⟨name⟩ ⟨definition⟩)

   where ⟨definition⟩ has the structure

   (⟨lambda-declaration⟩ ⟨arguments⟩ ⟨body⟩)

   A ⟨lambda-declaration⟩ is either a LAMBDA or an NLAMBDA. Functions written by the File Package to symbolic files take this form. Consider the example

```
(DEFINEQ
     (CUBIC
          (LAMBDA (X)
               (ITIMES X (ITIMES X X))
)))
```

2. (⟨name⟩ ⟨arguments⟩ ⟨body⟩)

   The function type is assumed to be LAMBDA. Most functions defined by type-in take this form. Consider the example

```
(DEFINEQ
     (CUBIC (X)
          (ITIMES X (ITIMES X X))
))
```

The appropriate LAMBDA expression is automatically created by DEFINEQ as the function definition is read. Thus, when you print the definition of CUBIC after typing it in, you will see a LAMBDA expression inserted into the definition.

The ⟨name⟩ field causes an atom to be created or updated with a function definition using a LAMBDA or NLAMBDA expression. The function definition is placed in the atom's function definition cell. ⟨arguments⟩ is either NIL or a list of atoms representing arguments to be used by the function. The ⟨body⟩ is one or more S-expressions that are inspected for proper form (see below) and comprise the specification of the function.

DEFINEQ works even if the function is broken, advised, or broken-in (see Chapters 20 and 21).

If time stamping is enabled (see Section 29.6.2), DEFINEQ and DEFINE stamp the definition with a comment consisting of your initials and the date when the function was defined.

8.2.4 The Effect of DFNFLG

DFNFLG determines how DEFINE treats the proposed definition:

DFNFLG is NIL

If DFNFLG is NIL and ⟨name⟩ already has a definition, DEFINE displays the message (⟨name⟩ REDEFINED). It saves the old definition, via SAVEDEF (see Section 17.5.7), on ⟨name⟩'s property list before redefining it.

Consider the following example:

```
←DFNFLG
NIL

←(DEFINEQ (CUBIC (X) (ITIMES X (ITIMES X X))))
(CUBIC)

←(DEFINEQ (CUBIC (X) (EXPT X 3)))
(CUBIC redefined)
(CUBIC)

←(GETPROP 'CUBIC 'EXPR)
(EXPR (LAMBDA (X) (* edited: "11-June-84 20:06")
(ITIMES X (ITIMES X X))))
```

DFNFLG is T

If DFNFLG is T, the function is simply redefined without the warning message.

Consider the following example if it was executed after the second expression in case 1:

```
←(SETQ DFNFLG T)
T

←(DEFINEQ (CUBIC (X) (EXPT X 3)))
(CUBIC)
```

DFNFLG is PROP
If DFNFLG is PROP or ALLPROP, the new definition is stored on ⟨name⟩'s property list under the property EXPR. However, it does not place any definition in the atom's function definition cell. Thus, attempting to get the definition of the function using GETD will return NIL. However, PP will work because it inspects both the function definition cell and the property list of the atom for a property named EXPR.

DFNFLG initially has the value NIL. It is reset by LOAD (see Section 17.9.1) when you load functions and variables from a file.

8.2.5 Alternative Defining Forms

INTERLISP/370 provides two alternative forms for defining functions: DE and DF. DE and DF define LAMBDA and NLAMBDA functions, respectively, in half-spread format. They take the following form

```
Function:       DE
                DF

# Arguments:    3-N

Arguments:      1) a function name, FN
                2) an argument list, ARGLST
                3-N) one or more S-expressions composing
                     the body of the function, EXPRS

Value:          The function name, if successful.
```

For example, we might define the function DISJOINT as

```
(DE disjoint (la lb)
    (COND
          ((NULL la) T)
          ((MEMBER (CAR la) lb) NIL)
          (T
               (disjoint (CDR la) lb))))
```

Basically, these functions provide a shorthand notation for defining functions that allow you to eliminate a few of the parentheses required by DEFINEQ. They have a form similar to DEFUN which is the defining function for the MACLISP family.

A Definition for DE
We might define DE as follows:

```
(DEFINEQ
     (de (NLAMBDA (x . y)
```

```
        (DEFINE
              (LIST
                    (LIST x
                          (CONS 'LAMBDA y))))
    ))
```

DF merely replaces the 'LAMBDA in the CONS expression by 'NLAMBDA to achieve the same effect.

8.3 RETRIEVING A FUNCTION DEFINITION

You may retrieve the definition of a function from its function definition cell by invoking **GETD**. GETD takes the format

```
Function:       GETD
# Arguments:    1
Argument:       1) the name of a function, FN
Value:          The definition of the function.
```

GETD returns the definition of a function if it exists; otherwise, NIL. For example,

```
(DEFINEQ
     (exchange (pair)
          (LIST (CADR pair) (CAR pair))
     ))

←(GETD 'exchange)
(LAMBDA (PAIR) (LIST (CADR pair) (CAR pair)))
```

If the function is compiled or a SUBR, GETD returns an encoded form representing the address of the first instruction of the function. For example (under the Fugue release of INTERLISP-D),

```
←(GETD 'getd)
{(CCODEP)}#1,161244
```

which is the encoded form of the dotted pair. CCODEP means that GETD is a compiled code pointer. #1 means that it takes one argument. 161244 is the memory address where the first executable instruction of that procedure is located.

Executing the same function under INTERLISP-10 returns the following form

```
←(GETD 'GETD)
(1 . {STACK}#11271)
```

which reflects the different memory allocation models used in the two implementations. Note that INTERLISP-10 returns the CONS of the function type with the address of the function.

Note that INTERLISP returns a pointer to the function. Two successive calls to GETD will return two different pointers. These pointers are not EQ. Rather, EQUAL or EQP must be used to compare them.

8.4 SETTING A FUNCTION DEFINITION

We have seen, in Section 8.2, that we can define a function using either DEFINE or DEFINEQ. The effect of these functions is to place the given definition in the function definition cell of the symbolic atom that was specified as the name of the function. INTERLISP provides another mechanism for inserting values into an atom's function definition cell. This is the function **PUTD** and its variations.

PUTD takes the form

```
Function:       PUTD
                PUTDQ
                PUTDQ?

# Arguments:    2

Arguments:      1) the name of a function, FN
                2) a definition for the function,
                   DEFINITION

Value:          The value of DEFINITION.
```

The first argument must be a literal atom; otherwise, PUTD generates an error message: ARG NOT LITATOM. If the second argument is not a list, PUTD generates an error message: ILLEGAL ARG. NIL is a valid value for the second argument. Unlike DEFINE or DEFINEQ, PUTD does not check the list to see if it is a valid definition, but merely stores it away. For example,

```
←(PUTD 'f-to-c
       '(LAMBDA (temperature)
              (QUOTIENT
                   (DIFFERENCE temperature 32.0)
                   1.8))))
(F-TO-C)
```

which defines the atom F-TO-C as a function that converts a temperature given in fahrenheit to an equivalent measure in celsius. Note that the definition of the function given to PUTD must include either a LAMBDA or NLAMBDA.

Note that PUTD is not sensitive to the value of DFNFLG, but places the value of DEFINITION directly in the function definition cell. Any contents of the function definition cell are replaced by the new definition.

8.4.1 Alternative Forms of PUTD

An alternative form of PUTD is **PUTDQ**. PUTDQ is an NLAMBDA version of PUTD that assumes that both of its arguments are literal values. Thus, we may define F-TO-C as follows:

```
←(PUTDQ f-to-c
        (LAMBDA (temperature)
              (QUOTIENT
                   (DIFFERENCE temperature 32.0)
                   1.8))))
(F-TO-C)
```

Another form of PUTD is **PUTDQ?**. PUTDQ? is an NLAMBDA form of PUTD that sets the value of the function definition cell of the first argument if and only if it is not defined. That is, it acts like PUTDQ. Otherwise, it returns NIL and does nothing.

A Definition for PUTDQ?
We might define PUTDQ? as follows:

```
(DEFINEQ
     (putdq?
          (NLAMBDA (fn definition)
               (AND (NULL (GETD fn))
                    (PUTD fn definition))
     )))
```

8.5 COPYING FUNCTION DEFINITIONS

Because function definitions are just lists or pointers to sequences of compiled code, we can copy a function definition from one atom to another. There are two reasons why you might want to perform this operation.

First, to simplify the definition of several similar functions. In this case, you define a (perhaps lengthy) function. Then, you copy its definition to another atom and edit the definition to produce a similar but different function.

Second, you may want to dynamically assign a function definition to an atom based on the current state of your program.

INTERLISP provides **MOVD** to copy function definitions between atoms. An alternative form, **MOVD?**, copies the function definition to the destination atom if and only if its function definition cell is NIL; otherwise, it does nothing.

The generic format for invoking these functions is

```
Function:      MOVD
               MOVD?
```

```
# Arguments:    3
Arguments:      1) a function name, FROMFN
                2) a function name, TOFN
                3) a copy flag, COPYFLAG
Value:          The name of the TOFN atom (MOVD/MOVD?);
                otherwise, NIL.
```

Note that MOVD works for EXPRs, compiled functions, and SUBRs. COPYFLAG, which is valid only for EXPRs, indicates that a copy of the function definition will be used if it is T. Otherwise, a pointer to the EXPR list is placed in the TO-atom's function definition cell. A new copy of the EXPR list allows us to edit the function definition as desired in case 1 above.

A Definition for MOVD

We might define MOVD as follows:

```
(DEFINEQ
    (movd (fromfn tofn copyflag)
        (PROG (newflag)
            (SETQ newflag
                (NULL (GETD tofn)))
            (PUTD tofn
                (COND
                    (copyflag
                        (COPY (GETD fromfn)))
                    (T
                        (GETD fromfn))))
            (RETURN tofn))
    ))
```

8.5.1 A MOVD Example

Suppose we can apply one of several disjunction operators to a pair of arguments. One way to code this is to assign the name of the disjunction function to an atom and use that atom as the argument of a SELECTQ statement. For example

```
←(SETQ disjunction-operator
      (QUOTE ⟨a-disjunction-function⟩))

←(SELECTQ disjunction-operator
          (disjunct1    (disjunct1 a b))
          (disjunct2    (disjunct2 a b))
              ...
          (disjunctN    (disjunctN a b)))
```

An alternative method would be to copy the function definition of the appropriate function to the function definition cell of DISJUNCTION-OPERATOR. Then, we could code DISJUNCTION-OPERATOR uniformly in our program. The current definition would be applied to the arguments whenever a statement of the form

```
(disjunction-operator a b)
```

is executed.

Consider the following definitions for possible disjunction functions:

```
(DEFINEQ
     (disjunct0 (a b)
          (COND
               ((ZEROP a) b)
               ((ZEROP b) a)
               (T 1))
     ))
(DEFINEQ
     (disjunct1 (a b)
          (MIN 1 (PLUS a b))
     ))
(DEFINEQ
     (disjunct2 (a b)
          (DIFFERENCE
               (PLUS a b)
               (TIMES a b))
     ))
(DEFINEQ
     (disjunct3 (a b)
          (MAX a b)
     ))
```

To set the current definition of DISJUNCTION-OPERATOR, we might use the following statement:

```
(MOVD
     (SELECTQ ⟨some condition⟩
          (0   (QUOTE disjunct0))
          (1   (QUOTE disjunct1))
          (2   (QUOTE disjunct2))
          (3   (QUOTE disjunct3)))
     (QUOTE disjunction-operator)
     T)
```

To make this method as general as possible, we could package this statement as a function that can be called from anywhere in the program.

8.6 FUNCTION PREDICATES

INTERLISP provides several predicates for testing the type of a function. These predicates correspond to the values returned by FNTYP as discussed in Section 8.1. Each predicate returns T if and only if its argument is one of the type indicated in the following table

Function Predicates

Predicate	Truth Condition
SUBRP	SUBR, FSUBR, SUBR*, FSUBR*
CCODEP	CEXPR, CFEXPR, CEXPR*, CFEXPR*
EXPRP	EXPR, FEXPR, EXPR*, FEXPR* also, if the argument has a list definition that does not begin with LAMBDA or NLAMBDA

The general format for invoking these predicates is

```
Function:       SUBRP
                CCODEP
                EXPRP

# Arguments:    1

Argument:       1) a function name or a function
                   definition, i.e., a list beginning with
                   LAMBDA or NLAMBDA
Value:          T, if the truth condition in the table
                above is met; NIL otherwise.
```

These predicates may be defined using FNTYP.
Consider the following examples:

```
←(EXPRP 'CREATE.NODE)
T

←(CCODEP 'DEFINEQ)
T

←(SUBRP 'CAR)
T
```

A Definition for SUBRP
We might define SUBRP as follows:

```
(DEFINEQ
     (subrp (a-function)
          (COND
               ((SELECTQ (FNTYP a-function)
                    ((SUBR FSUBR SUBR* FSUBR*)    T)
                    (NIL))))
     ))
```

The other functions are defined analogously.

8.7 ARGUMENT LIST FUNCTIONS

An INTERLISP function may or may not have arguments. When it does, it is sometimes useful to be able to obtain information about the arguments associated with the function. Variable bindings are determined by function type

1. If the function is a LAMBDA, each argument is bound to a value that results from enumerating and evaluating the expressions given in the argument list of the calling expression.
2. If the function is an NLAMBDA, the expressions in the calling expression are not evaluated, but are bound directly to the atoms in the argument list.
3. If ⟨arguments⟩ in the function definition is NIL, the function receives no arguments when it is called.
4. If ⟨arguments⟩ in the function definition is a single atom, the arguments in the calling expression are gathered into a list and assigned as the value of that atom. Expressions in the calling expression's argument list are evaluated or not depending on whether the function is a LAMBDA or NLAMBDA function.

INTERLISP provides several functions to determine the characteristics of arguments that are passed to a function.

8.7.1 Determining The Argument Type

You may need to determine the type of arguments a function expects in order to construct the argument list to be passed to that function. Usually, this case arises when you are interfacing with code written by another user. **ARGTYPE** takes one argument—the name of a function—and returns an integer that specifies the type of function that it is or NIL if it is not a function. It takes the following form:

```
Function:       ARGTYPE
# Arguments:    1
```

```
Argument:       1) a function name, FN
Value:          The argument type or NIL.
```

The value of ARGTYPE is specified by the following table:

Argument Types of Functions

Value	*Function Type*
0	Lambda-spread
1	Nlambda-spread
2	Lambda-nospread
3	Nlambda-nospread

Suppose that we have a function F1 that has been defined by another user, perhaps in a package that has been compiled. We do not know the type of the function nor do we care to visually inspect the code, if we could, to determine what the type of the function is. Nevertheless, we must use the function in our program. To do so properly, we have to construct an argument list to pass to the function. We might write a function in our own program called BUILD-ARGUMENT-LIST that constructs the proper argument list based on the type of function.

A skeletal definition of BUILD-ARGUMENT-LIST might appear as follows:

```
(DEFINEQ
    (build-argument-list (a-function an-argument-list)
        (SELECTQ (ARGTYPE a-function)
            (0   (make-ls-list argument-list))
            (1   (make-ns-list argument-list))
            (2   (make-ln-list argument-list))
            (3   (make-nn-list argument-list))
            (NIL NIL))
    ))
```

where each of the functions MAKE... construct the proper argument list for the appropriate function type.

To call the function F1, we would use APPLY:

```
(APPLY f1 (build-argument-list f1 argument-list))
```

Applying ARGTYPE directly to several standard INTERLISP functions, we see that it returns:

```
←(ARGTYPE 'DEFINEQ)
3
```

```
←(ARGTYPE 'ARGTYPE)
0
←(ARGTYPE 'PUTDQ)
1
←(ARGTYPE 'IPLUS)
2
```

8.7.2 Determining the Number of Arguments

You may also want to determine how many arguments a function requires when it is invoked. This knowledge is useful in constructing an argument list for a function. **NARGS** returns an integer that is the number of arguments in the function's argument list or NIL if its argument is not a function. It takes the form

```
Function:       NARGS
# Arguments:    1
Arguments:      1) a function name, FN
Value:          The number of arguments or NIL.
```

Its value is determined by:

1. NIL, if FN is not a function.
2. 0, if the argument list of FN's definition is NIL.
3. 1, if FN is a nospread function or has a single argument.
4. N, which is the actual number of arguments expected by FN.

Consider the following example:

```
←(NARGS (FUNCTION build-argument-list))
2
```

because BUILD-ARGUMENT-LIST requires two arguments.

```
←(NARGS 'RECORDACCESS)
5
←(NARGS 'LOGOUT)
0
```

NARGS uses EXPRP rather than FNTYP. Therefore, NARGS will work on S-expressions that are not functions but which are lists. For example,

```
←(NARGS '(do-test a-list an-operation))
2
```

where DO-TEST has not been defined as a function. However, it appears to take two arguments (if it were treated as a function) and so the terms A-LIST and AN-OPERATION are treated as arguments to the presumed function DO-TEST.

A Definition for NARGS

We might define NARGS as follows (after INTERLISP/370):

```
(DEFINEQ
    (nargs (fn)
        (COND
            ((EXPRP (SETQ fn (CGETD fn)))
                (*
                    If FN has a function definition
                    or an EXPR property. Extract the
                    argument list from the function
                    definition.
                )
                (SETQ fn (CADR fn))
                (COND
                    ((NULL fn)
                        (*
                            If the argument list
                            is NIL, return zero.
                        )
                        0)
                    ((ATOM fn)
                        (*
                            If the argument list
                            is an atom, its a
                            nospread function.
                        1)
                    ((NULL (CDR (LAST fn)))
                        (*
                            An argument list with
                            some number of atoms
                            representing
                            arguments.
                        )
                        (LENGTH fn))
                    (T
```

```
                    (*
                        The argument list has
                        a dotted pair at its
                        end.
                    )
                    (ADD1 (LENGTH fn)))))
            ((CCODEP fn)
                (*
                    The function is a compiled
                    function, so look at the linkage
                    to the function. This is machine
                    dependent.
                )
                (LOGAND
                    (LRSH
                        (MKN
                            (CAR (IPLUS 8
                                        (MKN fn))))
                        8)
                    255))
            (T
                (*
                    Functions defined as part of the
                    underlying virtual machine have
                    only one or two arguments.
                )
                (SELECTQ (FNTYP fn)
                    ((SUBR FSUBR) 2)
                    ((SUBR* FSUBR*)     1)
                    NIL)))
    ))
```

where CGETD is defined as

```
(DEFINEQ
    (cgetd (fn)
        (COND
            ((LITATOM fn)
                (GETD fn))
            (T fn))
    ))
```

CGETD merely distinguishes between an atom, which is assumed to be the name of the function, and an S-expression, which is assumed to be a LAMBDA/NLAMBDA form.

8.7.3 Obtaining the Argument List

Given a function that you did not define, but which you wish to invoke, you may determine the argument list by executing the function **ARGLIST**. ARGLIST takes the form

```
Function:       ARGLIST
# Arguments:    1
Argument:       1) a function name, FN
Value:          The arguments for the function.
```

ARGLIST returns a list of the arguments of the function which it takes as its argument. This function is particularly useful when you want to prompt the user for the values of arguments.

The value it returns depends on the function type and definition

1. NIL, if the function definition had NIL as its argument list specification.
2. An atom, if FN is a nospread function.
3. A list of atoms which are the names of the arguments that appeared in the function definition.
4. An error message, ARGS NOT AVAILABLE, if FN is not a function.

Consider the following examples:

```
←(ARGLIST 'BREAKIN)
(FN WHERE WHEN BRKCOMS)

←(ARGLIST 'ASKUSER)
(WAIT DEFAULT MESS KEYLST TYPEAHEAD LISPXPRNTFLG
OPTIONSLST FILE)
```

Certain functions in various INTERLISP implementations are hardwired via assembly language (INTERLISP-10/VAX/370) or microcode (INTERLISP-D). Thus, the actual names of the arguments are not preserved in the sysout. In this case, ARGLIST "manufactures" dummy argument names. For example,

```
←(ARGLIST 'LIST)
U

←(ARGLIST 'RPLACA)
(X Y)
```

If FN is a compiled function, the argument list must be reconstructed from the arguments. Thus, each call to ARGLIST will cause the construction of a new list.

If FN is an atom with an EXPR property whose value is a list beginning with LAMBDA or NLAMBDA, ARGLIST returns the CADR of GETD of the value. ARGLIST also works if it is given a list whose CAR is LAMBDA or NLAMBDA. For example,

```
←(ARGLIST '(LAMBDA (X) (ITIMES X X)))
(X)
```

8.7.4 Accessing the Arguments of a Nospread Function

A nospread function has a single atom as its argument list specification. Any expressions appearing after the function name in the calling expression are evaluated and their values are bundled into a list and bound to that single atom. To access individual arguments, you may use **ARG** which takes the form

```
Function:       ARG
# Arguments:    2
Arguments:      1) a variable name, VARX
                2) an index into the argument list, N
Value:          The value of the corresponding argument.
```

ARG is an NLAMBDA function. VARX is the name of the atom that appeared in the nospread function's definition. It may be any atom except NIL or T. It is *not* evaluated by ARG. N is an index into the argument list that specifies the number of the argument that you wish to retrieve. It is evaluated as follows:

1. If N is less than or equal to 0, ARG returns NOBIND.
2. If N is greater than the number of arguments, ARG returns NOBIND.
3. ARG returns the value of the corresponding argument.

Consider the following example (after the IRM):

```
(DEFINEQ
    (xplus
        (LAMBDA varx
            (PROG ((ARGNUM 0) (ARGVAL 0))
            loop
                (COND
                    ((EQ ARGNUM varx)
                        (RETURN ARGVAL)))
                (SETQ ARGNUM (ADD1 ARGNUM))
                (SETQ ARGVAL
                    (PLUS ARGVAL
                        (ARG varx ARGNUM)))
```

```
                    (GO loop))
    )))
```

Note that the variable VARX is bound to the number of arguments for a LAMBDA nospread function. When XPLUS is called, VARX is bound to 3. Thus, the test for terminating the loop succeeds. However, you should never reset the LAMBDA variable. Individual arguments may be reset via SETARG below.

Consider the following example:

```
←(XPLUS 10 30 50)
90
```

where we have the following correspondences:

```
←(ARG varx 1)
10
←(ARG varx 2)
30
←(ARG varx 3)
50
```

and the value of VARX is 3.

8.7.5 Setting the Arguments of a Nospread Function

SETARG allows you to set the arguments of a LAMBDA nospread function from within the function. It takes the form

```
Function:       SETARG
# Arguments:    3
Arguments:      1) a variable name, VARX
                2) an index, N
                3) a value to be set, VALUE
Value:          The new value.
```

SETARG is an NLAMBDA function. SETARG sets to VALUE the Nth argument of the LAMBDA nospread function whose argument list is given by the atom VARX. VARX is not evaluated, but N and VALUE are evaluated. It is primarily used to modify the argument list prior to processing it within the function.

8.8 FUNCTION EVALUATION

INTERLISP operates in a READ-EVAL-PRINT loop. Expressions are read from the primary input file, evaluated, and the results are printed on the primary output file. **EVAL** is the function which evaluates (e.g., "executes") the expression and returns its value. EVAL takes the form

```
Function:       EVAL
# Arguments:    1
Arguments:      1) an S-expression, EXPRESSION
Value:          The value, if any, produced by evaluating
                EXPRESSION.
```

EVAL takes one argument, EXPRESSION, which is an S-expression, that it gives to the intepreter for execution. Consider the following example:

```
←(SETQ ex1 '(ITIMES X X X))
(ITIMES X X X)
←(SETQ X 20)
20
←(EVAL ex1)
8000
←ex1
(ITIMES X X X)
```

EVAL is a LAMBDA function, so its argument is evaluated before being bound to EXPRESSION. EVAL is primarily used within functions where we construct an S-expression and then execute it. The following sections discuss some situations in which EVAL may be profitably used (some reference is made to concepts introduced in later chapters).

8.8.1 Updating a Database Variable

The File Package (see Chapter 17) uses a variable having the name ⟨file⟩VARS to represent a list of variables that are to be "saved" when a new version of the symbolic file is written. If your program processes several databases, you cannot hardwire the VARS variable into the code as some of the File Package functions expect. However, you may create your own updating function as follows:

```
(DEFINEQ
    (add.to.database.variables (a.variable)
        (SET current.database.vars
```

```
                (APPEND (EVAL current.database.vars)
                        (LIST a.variable)))
    ))
```

where CURRENT.DATABASE.VARS holds the name of the VARS variable.

Suppose we were maintaining a database about countries of the world. Assume that we needed to add the variable PUERTO-RICO to the USA database. When the database is loaded, we would set CURRENT.DATABASE.VARS to USAVARS. Then, adding Puerto-Rico would merely involve

```
←(add.to.database.variables 'puerto-rico)
(maryland puerto-rico)
```

Note that CURRENT.DATABASE.VARS has as its value the name of the file package variable. We evaluate this variable to obtain its value for updating.

8.8.2 A-list Evaluation

EVALA simulates a-list evaluation as it was performed in Lisp 1.5 [mcca72]. It takes the form

```
Function:       EVALA

# Arguments:    2

Arguments:      1) an expression, EXPRESSION
                2) a list of dotted pairs, ALST

Value:          The value produced by evaluating
                EXPRESSION.
```

EVALA spreads ALST on the stack. Each entry in ALST is a dotted pair consisting of a variable name and a value. EXPRESSION is evaluated using the "free" variables that appear on the stack as a result of spreading ALST. This form of evaluation was used in early LISP implementations. I recommend that you do not use it since its form and implementation are archaic.

8.8.3 Constant Evaluation

In some applications, you may need to specify functional arguments, but want them to evaluate to constants. INTERLISP provides functions that evaluate to the most frequently used constants. They take the form

```
Function:       NILL
                TRUE
                ZERO
```

```
# Arguments:    0
Arguments:      N/A
Value:          NIL, T, or 0, respectively.
```

These functions are nospread functions that always produce the specified constants.

8.9 FUNCTION APPLICATION

One of the nice characteristics of INTERLISP is the ability to define a function dynamically, and then use that function to accomplish some result. In general, this capability is not available in most other conventional programming languages. It is available because data structures and functions are just S-expressions in INTERLISP. So, to create a new function, we merely build an S-expression of its definition and then evaluate it. A problem arises in telling the function about its arguments; that is, in setting up the proper calling sequence because the function has not been formally defined to INTERLISP via DEFINE. To solve this problem, we can use **APPLY** which takes a function definition and a list of arguments. It takes the following format:

```
Function:       APPLY
# Arguments:    2
Arguments:      1) a function definition, FN
                2) a list of arguments, ARGS
Value:          The result of applying FN to ARGS.
```

FN may take several different forms:

1. It may be the literal name of a function:

```
←(APPLY 'PLUS '(5 6))
11
```

2. It may be a FUNCTION specification:

```
←(APPLY (FUNCTION TIMES) '(5 6))
30
```

3. It may be a LAMBDA or NLAMBDA definition:

```
←(APPLY '(LAMBDA (x y)
                 (CONS y x))
```

```
                    '(apples oranges))
          (oranges . apples)
```

APPLY does not evaluate the individual elements of ARGS. Rather, it simply calls FN with ARGS as the argument list. Thus, LAMBDA and NLAMBDA functions are treated exactly the same.

APPLY is a LAMBDA function, so it evaluates its arguments before it is called. Suppose I have a set of functions for computing weighted conjunctions of two numbers which represent evaluations of two conditions. CONJUNCT computes the proper evaluations given two numeric operands, but its operation may vary with the state of the program. We might define CONJUNCT as follows:

```
(DEFINEQ
     (conjunct (x y)
          (COND
               ((AND
                    (NUMBERP x)
                    (NUMBERP y))
                    (APPLY *conjunct-operator* x y))
               (T
                    (ERROR "Non-numeric Arguments")))
     ))
```

where *CONJUNCT-OPERATOR* is a global variable whose value is the name of a conjunction function.

We might define several conjunction functions as follows:

```
(DEFINEQ
     (conjunct1 (x y)
          (COND
               ((EQP x 1.0) y)
               ((EQP y 1.0) x)
               (T 0.0))
     ))

(DEFINEQ
     (conjunct2 (x y)
          (fmaximum 0.0 (PLUS x y (MINUS 1.0)))
     ))

(DEFINEQ
     (conjunct3 (x y)
          (PLUS 1.0 (TIMES x y))
     ))
```

Each function computes a different weighting of the variables X and Y. To select a function, we set its name as the value of *CONJUNCT-OPERATOR*.

```
←(SETQ *conjunct-operator* 'conjunct1)
conjunct1
←(conjunct 1.0 3.0)
3.0
←(SETQ *conjunct-operator* 'conjunct2)
conjunct2
←(conjunct 1.0 3.0)
3.0
←(SETQ *conjunct-operator* 'conjunct3)
conjunct3
←(conjunct 1.0 3.0)
4.0
```

8.9.1 APPLYing to an Indefinite Number of Arguments

Sometimes, the individual arguments to FN are produced separately in the calling program. To use these arguments in APPLY, you would have to invoke it in the following form

```
(APPLY ⟨FN⟩ (LIST ⟨arg1⟩ ⟨arg2⟩ ... ⟨argN⟩))
```

INTERLISP provides **APPLY*** as a shorthand notation for this form. It takes the format

```
Function:       APPLY*
# Arguments:    1-N
Arguments:      1) a function specification, FN
                2-N) arguments for FN
Value:          The result of applying Fn to ARGS[i]
```

APPLY* is a nospread function. It is useful where the arguments are calculated prior to applying the function. For example, you may have an expression of the form

```
(APPLY* (select-print-function) (select-print-arguments))
```

which determines which printing function to apply to the specified arguments.

8.10 REPETITIVE EXECUTION

Many expressions are executed several times in a program in succession. Usually, the expression maintains internal variables whose values change with each

iteration. Normally, you would have to set up a PROG loop to count the number of times that the expression is executed with a test to decide when you have iterated enough times. A shorthand notation for this mechanism is provided by **RPT**, which takes the form

```
Function:       RPT
# Arguments:    2
Arguments:      1) a repetition count, RPTN
                2) an expression, RPTF
Value:          The value resulting from the last
                evaluation of RPTF.
```

RPT executes the expression RPTF for RPTN times. RPTN is counted down as each evaluation is performed. RPTF may use the value of RPTN to determine what to do at any given iteration. If RPTN is less than or equal to zero, RPTF is not evaluated; RPT returns NIL.

An alternative form, **RPTQ**, is an NLAMBDA, nospread version of RPT. Its first argument, RPTN is evaluated to determine the number of iterations. The remaining N arguments are expressions which are not evaluated prior to calling RPTQ. At each repetition, RPTQ evaluates each of the RPTF[i] in succession. Its value is the result of the last evaluation of the last expression, RPTF[N].

Consider the following examples:

```
←(RPT 10 '(PRINT RPTN))
10
9
8
7
6
5
4
3
2
1
1    (the returned value)
```

Note that RPTN is accessible in the expression that is executed by RPT. We can initialize an array to a specific constant using RPT as follows:

```
←(SETQ A1 (ARRAY 5 5))
{ARRAYP}#546261
```

```
←(RPT (ARRAYSIZE A1) '(SETA A1 RPTN (ITIMES RPTN 100)))
100
```

which is the last value assigned because RPT counts down from the upper limit. Inspecting the value of A1[3], we see that it is 300:

```
←(ELT A1 3)
300
```

A Definition for RPT
We might define RPT as follows:

```
(DEFINEQ
     (rpt (rptn rptf)
          (PROG (rptv)
          loop
                (COND
                    ((IGREATERP rptn 0)
                         (SETQ rptv (EVAL rptf))
                         (SETQ rptn (SUB1 rptn))
                         (GO loop))
                    (T
                         (RETURN rptv))))
     ))
```

8.11 GENERATORS

Whenever you call a function in INTERLISP, it creates a stack frame (see Chapter 30) that exists for the duration of the function's execution. When the function exits, the stack frame is released and all information concerning that activation is forgotten. If the function's task is to compute a series of values, you must

1. Compute all values and store them as a global variable. Functions external to the function that computed these values are responsible for accessing the individual values. Or,
2. Retain state information about the function in a global variable so that, at each invocation, the function can reestablish its previous state before computing the next value.

The former method may be time-consuming if a large number of values must be computed. Moreover, if the next value to be computed depends upon some massaging of the previous value by an external function, it may not be possible to compute all the values at one time. The latter method introduces substantial complexity into the function which makes it difficult to comprehend as well as as leading to possible errors due to wrong implementation.

Generators are a mechanism for circumventing this problem. A generator is a function that retains state information (e.g., a history of its previous invocations).

8.11.1 Initializing a Generator

GENERATOR is an NLAMBDA function that initializes a generator. It takes the format

```
Function:       GENERATOR
# Arguments:    2
Arguments:      1) an expression, FORM
                2) a generator handle, COMPVAR
Value:          A generator handle.
```

GENERATOR creates a stack frame for FORM so that it may be called repeatedly. it returns a *generator handle* which is a pair of stack pointers.

Consider the following example taken from the IRM [irm83]:

```
←(DEFINEQ
    (LISTGEN (lst)
        (IF lst
         THEN (PRODUCE (CAR lst))
              (LISTGEN (CDR lst)))
    ))
(LISTGEN)
```

LISTGEN will produce elements of the list one at a time.

```
←(SETQ GR (GENERATOR (LISTGEN '(A B C))))
(#1,13442/#0 . #1,13444/GENERATOR)
```

Each time GENERATE is called, it invokes PRODUCE to return the next value.

```
←(GENERATE GR)
A
←(GENERATE GR)
B
←(GENERATE GR)
C
←(GENERATE GR)
(#1,1342/#0 . #1,13444/#0)
```

When you have exhausted the input list, GENERATE returns the generator handle to indicate termination.

```
←(GENERATE GR)
STACK POINTER HAS BEEN RELEASED
#1,13444/#0
```

PRODUCE and **GENERATE** take the following forms

```
Function:       PRODUCE
# Arguments:    1
Argument:       1) a value to be returned, VALUE
Value:          The value.
```

PRODUCE is used from within a generator to return a value each time the generator is called.

```
Function:       GENERATE
# Arguments:    2
Arguments:      1) a generator handle, HANDLE
                2) a value, VALUE
Value:          The value to be returned.
```

GENERATE restarts the generator given by HANDLE. VALUE is returned as the value of the PRODUCE which last suspended the operation of the generator. When the generator runs out of values (if it does), GENERATE returns the value of HANDLE itself.

8.12 MACROS

When a function is called in INTERLISP, a new stack frame is created to establish the environment in which the function will execute. Since INTERLISP does not distinguish between large and small functions, every function experiences this overhead. Sometimes, we need to parametrize very small pieces of code to do simple but diverse operations. Coding these code skeletons as fucntions forces us to pay the price of the function call.

Macros allow us to create code skeletons which are expanded in-line to produce efficient code. Macros are parametrizable, but invoking them does not cause a new stack frame to be created. They are akin to the in-line functions allowed in FORTRAN. Basically, executing a macro results in the creation of another piece of code which is then evaluated to produce a result. Thus, executing a macro is a two-step process: macro expansion and resultant code evaluation.

Any literal atom may have a macro definition, just as it has a function definition cell. The macro definition is stored on the atom's property list as a list. The property name under which the definition is stored depends on the implementation that you are using. Current property names include

```
MACRO          implementation independent macros
10MACRO        macros for INTERLISP-10
DMACRO         macros for INTERLISP-D
VAXMACRO       macros for INTERLISP-VAX
```

8.12.1 Defining Macros

A macro is defined by placing a value for a macro property name on the property list of a literal atom. Macro definitions can take several forms:

1. Lambda or Nlambda forms

   When functions are compiled, they generate code that includes the overhead necessary to create a stack frame. You may force a function to compile open (e.g., generate in-line code) by placing a macro definition with LAMBDA or NLAMBDA on the atom's property list. The definition takes the usual form of a LAMBDA or NLAMBDA S-expression.

   Consider the code for REPLACADD which has the definition

```
(DEFINEQ
     (replacaddd (x y)
          (RPLACA (LAST x) y)
          y))
```

   and a test function, TEST1, defined as

```
(DEFINEQ (TEST1 (X Y) (REPLACADDD X Y)))
```

   We may put the macro definition for REPLACADDD on its property list using

```
(PUTPROP 'REPLACADDD 'MACRO (GETD 'REPLACADDD))
```

   Now, compiling TEST1, we see the following code (under INTERLISP-10):

```
(TEST1 (x y) NIL)
     (ENTERF 2 0 0 0)
-1   (PUSHV x 0)
     (PUSHV y 1)
     (LAM
```

```
2           (RET)
            (LDV y 4)
            (VCLL y 5 RPLACA 2)
            (PUSHP)
            (VCLL x 4 LAST 1)
            (J 2)
            (BIND (x y) 2 0 2 2 NIL NIL T 2)
      )
      (RET)
```

But, if we remove the macro definition from REPLACADDD's property list via

```
(REMPROP 'REPLACADDD 'MACRO)
```

and recompile TEST1, we see the following code:

```
(TEST1 (x y) NIL)
      (ENTERF 2 0 0 0)
-1    (PUSHV x 0)
      (VCLL y 1 REPLACADDD 2)
      (RET)
```

In the case of the macro definition, the code for REPLACADDD was compiled in-line to the definition of TEST1 whereas in the second case, a function call was generated.

2. Substitution Macros

A substitution macro takes one of two forms:

```
(NIL ⟨expression⟩)
(⟨list⟩ ⟨expression⟩)
```

Each argument in the S-expression being evaluated or compiled is substituted for the corresponding atom in ⟨list⟩. The resulting expression is used instead of the form. Consider the example given in the IRM [irm83]

```
←(PUTPROP 'ADDONE 'MACRO '((X) (IPLUS X 1)))
((X) (IPLUS X 1))
←(SETQ y '(5 6))
(5 6)
←(EXPANDMACRO (ADDONE (CAR Y)))
6
```

3. T Macros

When a macro definition has the value T, the compiler ignores the macro definition and compiles the function definition instead. Normally, this will be used with atoms that have both a MACRO property and a specific implementation MACRO property, such as 10MACRO. The MACRO property has the macro definition. If you want to run the function on INTERLISP but the macro definition is valid only for INTERLISP-D, setting the DMACRO property to T instructs the compiler to ignore it when compiling the expression on INTERLISP-10 systems.

4. Synonym Macros

You may instruct the compiler to treat one function as if it were another when compiling expressions by specifying MACRO properties of the following form:

```
(= . ⟨other-function-name⟩)
```

For example, INTERLISP-D compiles many of the functions beginning with F (for "fast"—see the IRM) exactly like their interpreted brethren (because of the way they are implemented in INTERLISP-D). Thus, FRPLACAs are treated exactly like RPLACAs. However, this approach does not work with any of the other implementations because they compile into machine language.

These types of macros may be used to define synonyms for functions. Suppose you frequently misspell the name of a function in a file. Rather than searching the file to replace all of the misspellings, you might place a synonym macro on the misspelling which specifies the correct function to be used in compiling the file.

5. Computational Macros

A macro definition beginning with a literal atom other than those above causes the S-expression value to be evaluated or compiled in place of the form. This form is specified by

```
(⟨literal-atom⟩ ⟨expression⟩)
```

The literal atom receives the argument list to the macro.

The IRM suggests that LIST could be compiled using the alternative form

```
(X
    (LIST 'CONS
```

```
        (CAR X)
        (AND (CDR X)
             (CONS 'LIST
                   (CDR X)))))
```

Thus, the expression (LIST 'MOSES 'ABRAHAM 'ISHMAEL) would be compiled as

```
(CONS MOSES
      (CONS ABRAHAM
            (CONS ISHMAEL NIL)))
```

In this case, the macro expansion contains a recursive expression that allows it to deal with any number of arguments to LIST.

For example, giving this macro property to TEST2:

```
←(PUTPROP 'TEST2
          'MACRO
          '(X
              (LIST 'CONS
                    (CAR X)
                    (AND (CDR X)
                         (CONS 'LIST
                               (CDR X))))))
(X (LIST 'CONS (CAR X) (AND (CDR X) (CONS 'LIST (CDR
X)))))

←(TEST2 'MOSES 'ABRAHAM 'ISHMAEL)
(MOSES ABRAHAM ISHMAEL)

←(TEST2 10 20 30 40 50)
(10 20 30 40 50)
```

where X has taken the value (10 20 30 40 50) prior to expanding the macro definition.

If the literal atom evaluates to the litatom IGNOREMACRO, the macro is ignored and the compilation or evaluation proceeds as if there were no macro definition.

8.12.2 Expansion of Macros

Literal atoms may have both a function definition and a macro definition. When the interpreter evaluates an expression, it inspects the CAR of the expression. If the CAR of the expression has a function definition, that will be used (via a function call) to evaluate the expression. If it has a macro definition, then the expansion of the macro will be used to evaluate the expression.

The reverse is true during compilation. The macro definition is checked first. If it exists, it is used to generate the code for evaluating the expression, subject to the constraints mentioned above. If there is no macro definition, then the function definition is used.

The IRM suggests that you may want to use a function definition that has a lot of error handling and debugging code for interpreted expressions during system development, but replace this by a fast in-line macro expansion when you compile the same expression (without the debugging code).

Interpreted macros are implemented by the function MACROTRAN (for macro translation) which is an entry on DWIMUSERFORMS. If DWIM (see Chapter 22) is not enabled, MACROTRAN will not work.

MACROTRAN is called if the CAR of an expression is undefined; that is, does not have a function definition. If the CAR of the expression has a macro definition, it is expanded and the result is evaluated in place of the original expression. The value of the expansion (an expression itself) is saved by CLISPTRAN (see Chapter 23) in CLISPARRAY so that the expansion need only be performed once. On subsequent findings, the expansion is retrieved from CLISPARRAY without invoking MACROTRAN.

EXPANDMACRO is a function that is used to expand the macro definition of a literal atom and evaluate it during interpretation. It takes the form

```
Function:       EXPANDMACRO
# Arguments:    2
Arguments:      1) an expression, EXPRESSION
                2) a quiet flag, QUIETFLAG
Value:          The result of expanding the macro
                definition and evaluating it.
```

The CAR of EXPRESSION must have a macro definition. This is expanded by EXPANDMACRO and evaluated. The result will be prettyprinted, unless QUIETFLAG is T, whence it is just returned.

Consider the definition for TEST2 as demonstrated in the following example

```
←(EXPANDMACRO '(TEST2 MATTHEW))
(CONS MATTHEW NIL)
```

8.12.3 A Function for Defining Macros

Other LISP dialects provide functions for defining macros. For example, FranzLisp provides the function DEFMACRO which takes a form similar to a function definition. We might define DEFMACRO as follows

```
←(DEFINEQ
    (defmacro
        (NLAMBDA definition
            (PUTPROP (CAR definition)
                     'MACRO
                     (CDR definition))
            (CAR definition)
    )))
(DEFMACRO)

←(DEFMACRO NEQ (A B) (NOT (EQ A B)))
NEQ

←(GETPROP 'NEQ 'MACRO)
((A B) (NOT (EQ A B)))

←(EXPANDMACRO '(NEQ X Y))
(NOT (EQ X Y))

←(EXPANDMACRO (NEQ X Y))
T
```

with X set to ABC and Y set to DFE.

With this function, we may define two simple macros that treat a list like a stack. PUSH and POP have the following definitions

```
←(DEFMACRO PUSH (VALUE STACK)
                (SETQ STACK (CONS VALUE STACK)))
PUSH

←(DEFMACRO POP (STACK)
               (PROG1
                   (CAR STACK)
                   (SETQ STACK (CDR STACK))))
POP

←(SETQ STACK NIL)
NIL

←(SETQ VALUE 'X)
X

←(PUSH VALUE STACK)
(X)

←(POP STACK)
X

←STACK
NIL
```

9

Atom Manipulation

As we mentioned in Section 2.1, a literal atom is an indivisible unit of storage that is allocated by INTERLISP at the user's request. Atoms have names, the **PRIN1-NAME** (PNAME), that are printed by INTERLISP. Two atoms that have the same PNAME have identical addresses in memory. Atoms are unique. The PNAME is defined as the sequence of characters that will be displayed by PRIN1.

An atom may be characterized by a property list, a value, a function definition, and a PNAME. Property lists and their manipulation are discussed in Chapter 7. Function definitions and their manipulation are discussed in Chapter 8. This chapter discusses the creation and manipulation of atoms, and the printing of their names and values.

The *value cell* of an atom is a set of memory locations that are allocated and assigned to an atom when it is given a value. An atom may be given a value in three ways:

1. Referencing it as a variable via SETQ
2. Specifying it as a function parameter
3. Specifying it as a PROG variable

An atom that has not been assigned a value has the atom NOBIND placed in its value cell. Thus, a value cell contains either an explicit value (one that fits into a computer word, such as an integer) or a pointer to a set of memory locations that contain the atom's value.

Value cells cannot be directly referenced by the user. They must be accessed by referencing the atom as a variable.

The *PNAME* of an atom is the collection of characters that represents the name of the atom as it appears when entered by the user or printed by the system. The length of the PNAME is implementation dependent. For example,

```
(SETQ maryland NIL)
```

will create the atom MARYLAND if it did not previously exist in the system. The sequence of characters MARYLAND is the PNAME of the atom. The user references the value of this atom by typing MARYLAND to the top-level READ-EVAL-PRINT loop of INTERLISP. For example,

```
←maryland
NIL
```

9.1 RULES FOR ATOM NAMES

The length of the PNAME of a literal atom depends on the implementation. Currently, this length is 127 for INTERLISP-10, INTERLISP/VAX, and INTERLISP/370, and 255 characters for INTERLISP-D. Attempting to create a literal atom with a PNAME whose length exceeds these limits causes the error "ATOM TOO LONG" to be generated.

Literal atom names are any sequence of characters that

1. Cannot be interpreted as a number
2. Are delimited by one of the following syntactic characters:

```
space
EOL         end-of-line
⟨LF⟩        line feed
(           left parenthesis
)           right parenthesis
"           double quote
[           left bracket
]           right bracket
```

However, any character may be included in a literal atom name if it is preceded by an ⟨ESC⟩ character, **%**. For example, we may create a rather unusual atom name via the following expression

```
←(SETQ A%(BC%)D NIL)
NIL
```

which creates the atom A(BC)D. To ask INTERLISP to print its value, we must explicitly type the escape character before the left and right parentheses to indicate they are part of the atom's name. Thus, we must enter

```
←A%(BC%)D
NIL
```

A PNAME is a collection of characters that are output when a pointer is printed using PRIN1 (see Section 15.1.1) or PRINT (see Section 15.1.3). This is

often referred to as the PRIN1-PNAME. PRIN1 does not print escape characters that might occur in the name.

An alternative function, **PRIN2** (see Section 15.1.2), prints all escape characters in the pointer's name. PRIN2's action may be modified by the action of a read table.

The PNAME of an integer depends on the value of **RADIX** (see Section 15.4.4). Integers are always printed by PRIN1 using the current value of RADIX.

The number of characters permitted in a literal atom name allows you to be both expressive and creative. It is hard to underestimate the value of using good mnemonic names, and yet, you will find many INTERLISP programs that reflect the influence of FORTRAN where only six characters were permitted.

Good names, particularly when they reflect the usage of the variable, make a program easier to read. You may argue that you have to type in too many characters time after time, which is a tedious process. True, but given the tendency of INTERLISP programmers to fail to document their functions, lengthy names may help you remember what a function did weeks or months later.

I have found that it helps me read my programs by breaking up the literal atom names with periods (.) or dashes (-). The latter works most of the time, but CLISP (see Chapter 23) does have a tendency to interpret such names as the subtraction of two variables. You might also consider using an underscore (_), but I have found that some printers do not display that character. Rather, they substitute a space which often leaves you wondering whether you are looking at two atoms or a single atom with an unprintable underscore.

Consider some of the following atom names:

```
candidate.goals
deduce.plan.list
lexical.scanner
sentence.scanner
number.of.characters
```

It is unfortunate that most of the atom and function names used in INTERLISP do not follow this philosophy. You should contrast this with the readability of many ZetaLisp programs which use names broken into manageable segments as suggested above.

9.2 CREATING ATOMS

We have mentioned that atoms are normally created in three ways:

1. via SETQ
2. as a function parameter
3. as a PROG variable

In each of these cases, the name of the atom is specified by you when you execute an INTERLISP statement.

INTERLISP allows you to create atoms using three other functions: GENSYM, PACK, and MKATOM.

9.2.1 GENSYM: Generating a Symbol

You may create an atom using **GENSYM**, which takes the form

```
Function:       GENSYM
# Arguments:    1
Argument:       1) a character sequence, CHAR
Value:          A unique (usually) literal atom.
```

GENSYM appends up to four digits to the character sequence to create the atom name. The form of the atom name is

⟨character sequence⟩⟨digits⟩

If the ⟨character sequence⟩ is NIL, INTERLISP uses the default character A.

INTERLISP maintains an internal counter that is initialized to 10000. The current value of GENNUM, a system variable, represents the value that was used to create the last atom. GENNUM is incremented each time GENSYM is executed. Consider the following examples:

```
←GENNUM
10000
←(GENSYM)
A0001
←(GENSYM)
A0002
←(GENSYM 'x)
X0003
←(GENSYM 'help)
HELP4
```

Note that GENSYM suppresses the appropriate number of leading digits depending on the number of characters in its argument. Thus, when given the argument HELP above, GENSYM suppresses all leading zeroes, and merely appends the nonzero digits. This may cause a problem if you use character se-

quences with four characters in them. Consider the following expression (A CLISP expression) and its results:

```
←(FOR I FROM 1 TO 11 DO (PRINT (GENSYM 'HELP)))
HELP1
HELP2
HELP3
HELP4
HELP5
HELP6
HELP7
HELP8
HELP9
HELP0
HELP1
```

Note that the last atom name generated is not unique because GENSYM suppresses leading digits. Caution should be exercised when using GENYSM with character sequences larger than two or three characters if you expect to create many atoms.

If the character sequence is five characters in length, GENSYM merely returns that character sequence. For example,

```
←GENNUM
10011
←(GENSYM 'TRIAL)
TRIAL
```

but note that GENNUM is still incremented by one:

```
←GENNUM
10012
```

If the sequence is longer than five characters, GENSYM generates an error:

```
←(GENSYM 'PROCRASTINATION)
ILLEGAL ARG
PROCRASTINATION
```

A Definition of GENSYM

One possible definition of GENSYM might be as follows:

```
(DEFINEQ
   (gensym (a-string)
```

```
(PROG (atom-prefix)
      (SETQ atom-prefix a-string)
      (COND
         ((NULL atom-prefix)
              (*
                   If no character sequence is
                   specified, use the default.
              )
              (SETQ atom-prefix 'A))
         ((GREATERP (LENGTH (UNPACK atom-prefix))
                    5)
              (*
                   Generate an error if the
                   character sequence length is
                   greater than 5 characters.
              )
              (PRINT "ILLEGAL ARG")
              (PRINT atom-prefix)
              (RETURN)))
         (*
              Generate the atom name and increment
              GENNUM. Note the use of PROG1.
         )
        (RETURN
          (PROG1
            (SET
              (PACK
                 (APPEND
                    (UNPACK atom-prefix)
                    (LASTN (UNPACK GENNUM)
                           (DIFFERENCE 5
                              (LENGTH
                                 (UNPACK atom-prefix)))
                    )))
              NIL)
            (SETQ GENNUM (ADD1 GENNUM)))))
))
```

INTERLISP does not guarantee that the atom it creates will be unique. This situation may arise if you create atoms having the form specified above. Suppose the current value of GENNUM is 10056. Further, suppose you have previously created an atom J0057. If you execute

```
←(GENSYM 'j)
J0057
```

INTERLISP does not create a new atom J0057 because one alreay exists by that name. Assignment of a value to what you expected to be a new atom may, in fact, overwrite the value of the existing atom (possibly with disastrous consequences). Good programming practice dictates that you do not use atom names of this form in your programs.

It is permissible to reset the value of GENNUM within your program, but you do so at some risk. In fact, if you follow the advice regarding the specification of atom names given above, you should never run the risk of conflict between atom names created by GENSYM and those that you have created.

9.2.2 MKATOM: Creating Atoms from Strings

When executing programs, you may want to create atoms having names that are dependent on data read in or created by the program. For example, suppose you have a program that maintains a catalog of books by author. Entries in the catalog are represented by atoms constructed from the author's name. New entries in the catalog are made by reading the author's name and manipulating it to create a new atom.

We can create a new atom from the string representing the author's name using **MKATOM**, which takes the following form

```
Function:       MKATOM
# Arguments:    1
Arguments:      1) an expression, EXPRESSION
Value:          A new atom whose pname is the characters
                comprising the string.
```

If the value of EXPRESSION is not a string, INTERLISP applies MKSTRING to it. If this succeeds, MKATOM creates an atom, if one does not already exist, with the PNAME given by the value of the string.

Consider the following examples:

```
←(SETQ z "NOW IS THE TIME FOR ALL GOOD MEN")
"NOW IS THE TIME FOR ALL GOOD MEN"

←(MKATOM z)
NOW% IS% THE% TIME% FOR% ALL% GOOD% MEN
```

which is an atom with 32 characters in its name. The %s indicate that the spaces are valid characters in the name of the atom.

```
←(SETQ x (LIST 'julius 'caesar))
(julius caesar)

←(MKATOM x)
%(JULIUS% CAESAR%)
```

Note that the parentheses have been included as valid symbols in the name of the atom. This occurs because the result of evaluating (LIST 'JULIUS 'CAESAR) is (JULIUS CAESAR) which is then given to MKSTRING. If the string has a length greater than that allowed for an atom name in the particular implementation, the error "ATOM TOO LONG" will be generated.

```
←(SETQ x 3.141592)
3.141592
←(MKATOM x)
3.141592
←(MKATOM "3.141592")
3.141592
```

Numbers are literal atoms, by definition. No literal atom can have the PNAME of a number except the number itself.

A Definition for MKATOM
We can define MKATOM as follows:

```
(DEFINEQ
     (mkatom (a-string)
          (PROG (the-string)
               (SETQ the-string a-string)
               (COND
                    ((NOT (STRINGP a-string))
                         (SETQ the-string
                              (MKSTRING a-string))))
               (RETURN
                    (SET
                         (CAR
                              (PACK
                                   (UNPACK the-string)))
                         NIL)))
     ))
```

9.2.3 Making an Atom from a Substring

Given a string, you may make an atom name from a substring of the string using **SUBATOM**, which takes the form

```
Function:       SUBATOM
# Arguments:    3
```

```
Arguments:      1) an expression, EXPRESSION
                2) a staring index, START
                3) an ending index, END
Value:          A literal atom name created from the
                substring extracted from the value of
                EXPRESSION.
```

Usually, START and END are positive and characters are counted from the beginning of the string. If either START or END is negative, then that index is counted from the end of the string.

If the value of EXPRESSION is not a string, MKSTRING is applied to it to create a string to which SUBSTRING is applied.

Consider the following examples:

```
←(SETQ string "Washington was the father of his
country")
"Washington was the father of his country"
```

From this string we might extract a few substrings which become atoms in a database we are building from strings that are read into our program. For example, the key figure in this string is WASHINGTON, so we can extract his name as an atom via

```
←(SUBATOM string 1 10)
WASHINGTON
←(SUBATOM string 20 25)
FATHER
```

If START or END is negative, it indicates a position counting backwards from the end of the string. Thus, we could extract FATHER as follows

```
←(SUBATOM string -21 -16)
FATHER
```

Note that if the value of the substring could be interpreted as a number, then the value of SUBATOM would be the corresponding number. Consider the following example:

```
←(SUBATOM "July 4, 1776" 9 -1)
1776
```

A Definition for SUBATOM

We might define SUBATOM to be

```
(DEFINEQ
     (subatom (expression start end)
          (MKATOM (SUBSTRING expression start end))
     ))
```

9.3 PACKING AND UNPACKING ATOMS

As we mentioned, an atom is an indivisible unit. But the name of an atom is merely a sequence of characters which, themselves, are indivisible units. We can break an atom name up into its constituent parts using the function UNPACK (sometimes called EXPLODE in other LISP systems). Conversely, we create atom names from a sequence of characters (or character codes) using the function PACK (sometimes called IMPLODE in other LISP systems).

PACK and UNPACK allow you to create atom names which represent both similarity and diversity. For example, the first N characters of a name could be the same indicating a similarity of purpose of the names, while the remaining characters are different indicating the different purpose of each of the atoms.

9.3.1 Packing Atoms

PACK concatenates any number of individual atom names into a single atom name. PACK takes the following form

```
Function:       PACK
                PACKC

# Arguments:    1

Argument:       1) a list of atoms, LST

Value:          An atom.
```

In fact, PACK really concatenates the PNAMEs of the individual atoms in LST to form the PNAME of the result. If the argument is not a list, PACK generates an error message: ILLEGAL ARG. Consider the following examples:

```
←(PACK 'A)
ILLEGAL ARG
A

←(PACK '(r e a g a n))
REAGAN

←(PACK '(1 3 . 4))
13.4

←(PACK '(1 E -3))
.001
```

Note that if PACK produces the PNAME of a number as its result, when that PNAME is printed, it will appear correctly formatted by notation and radix.

If the length of the resulting atom name is longer than the maximum allowable atom name for your implementation, INTERLISP generates the error "ATOM TOO LONG".

A nospread version of PACK, **PACK***, takes an indefinite number of arguments but avoids the CONSes required to form the result. It takes the form

```
Function:       PACK*
# Arguments:    1-N
Arguments:      1-N) LISP objects
Value:          The concatenated result of the arguments
                as an atom's pname, if valid.
```

PACK* is a nospread function. Consider the following examples:

```
←(PACK* 'A 'Q 'U 'I 'N 'A 'S)
AQUINAS
←(PACK* 1 '. 7 8 2)
1.782
```

Another variant, **PACKC**, takes a list of character codes and returns the alphanumeric equivalent of the PNAME. For example,

```
←(PACKC '(45 67 89))
-?y
```

9.3.2 Unpacking Atoms

The converse function allows you to **UNPACK** the PNAME of an atom. Its value is a list of atoms corresponding to the PNAMEs of the characters comprising the PNAME of the argument. The generic format of UNPACK is

```
Function:       UNPACK
# Arguments:    1-3
Arguments:      1) an atom or string, ATM
                2) a flag, FLAG
                3) a read table, RDTBL
Value:          A list of atoms that comprise the PNAME of
                ATM.
```

Consider the following examples:

```
←(UNPACK '(washington))
(w a s h i n g t o n)

←(UNPACK 3.141592)
(3 %. 1 4 1 5 9 2)

←(UNPACK "BRIGADOON")
(B R I G A D O O N)
```

If FLAG is NIL, UNPACK produces a list of atoms corrsponding to the PRIN1-PNAME of the first argument. However, if FLAG has the value T, then the result corresponds to the PRIN2-PNAME of the first argument. The interpretation of the PRIN2-PNAME is modified by the RDTBL (see Section 14.4).

```
←(UNPACK '(WASHINGTON) T)
(%( W A S H I N G T O N %))

←(UNPACK "BRIGADOON" T)
(%'' B R I G A D O O N %")

←(UNPACK 3.141592 T)
(3 %. 1 4 1 5 9 2)
```

Note that UNPACK will execute N CONSes where N is the number of characters in the argument to be unpacked.

9.3.3 Using PACK and UNPACK

The primary use of UNPACK is to produce a list of atoms that may be manipulated by other functions.

Suppose that you have a program that processes an input file to produce an output file. You prompt the user for the basic file name. Using this name, you generate an input file name that has the proper extension to distinguish the file types according to function.

We can construct the function MAKE-INPUT-TEXT-FILE-NAME using PACK and UNPACK. This function assumes that the input file type is ".text"

```
(DEFINEQ
    (make-input-text-file-name (file-name)
        (PROG (input-file-name temp-file-name)
            (SETQ temp-file-name (UNPACK file-name))
            (SETQ input-file-name NIL)
        loop
            (COND
                ((EQUAL (CAR temp-file-name) '.)
                    (GO exit))
```

```
                (T
                     (TCONC input-file-name
                            (CAR temp-file-name))))
           (SETQ temp-file-name
                 (CDR temp-file-name))
           (AND
                (NULL temp-file-name)
                (GO exit))
           (GO loop)
      exit
           (RETURN
                (PACK
                     input-file-name
                     (UNPACK '.text))))
))
```

And, when we execute this function

```
←(MAKE-INPUT-TEXT-FILE-NAME 'MONDALE-CAMPAIGN)
MONDALE-CAMPAIGN.TEXT
```

9.4 CHARACTER CONVERSION

Converting between the numeric equivalent of a character and its PNAME format is a useful function. There are two functions to accomplish this: CHCON and CHARACTER.

9.4.1 CHCON: Converting to a Number

CHCON converts the PRIN1-PNAME equivalent of an atom to a list of its numeric equivalents. This function is dependent on the character code used by the machine on which INTERLISP is implemented. Consider the following example (assuming EBCDIC):

```
←(CHCON 'tungsten)
(163 164 149 135 162 163 133 149)
```

The generic format for calling CHCON is

```
Function:       CHCON
# Arguments:    1-3
Arguments:      1) an atom or string, ATM
                2) a flag, FLAG
                3) a read table, RDTBL
```

Value:	A list of the numeric equivalents of the characters comprising the PNAME of the first argument.

If FLAG has a value of T, the PRIN2-PNAME will be used instead of the PRIN1-PNAME. If the read table is non-NIL (see Section 14.4), it is used to interpret the characters in PRIN2-PNAME.

An alternative form, **CHCON1**, returns the character code of the first character of the atom or string. CHCON1 does not use either the PRIN2-PNAME or the read table as CHCON does. It takes the form

Function:	CHCON1
# Arguments:	1
Argument:	1) an atom or string, ATM
Value:	The character code of the first character of the PNAME of ATM.

Consider the following example:

```
←(CHCON1 'tungsten)
163
```

A Definition for CHCON1

We might define CHCON1 as follows:

```
(DEFINEQ
     (chcon1 (atm)
          (NTHCHARCODE atm 1)
     ))
```

9.4.2 CHARACTER: Converting to the PNAME Equivalent

CHARACTER takes a single character code as its argument and returns the PNAME equivalent. It takes the form

Function:	CHARACTER
# Arguments:	1
Argument:	1) a character code, CC
Value:	The atom with the corresponding character as its PNAME.

Consider the following examples:

```
←(CHARACTER 149)
N
←(CHARACTER 202)
J
←(CHARACTER 32)
%⟨space⟩              because 32 is the code for a blank!
←(CHARACTER 13)
%                     because 13 is a line-feed.
```

You should experiment with some of the non-printing characters to determine their behavior when the corresponding character codes are given to CHARACTER at the top level.

9.4.3 Character Code Structures

CHARCODE allows you to convert all the elements of an S-expression to character codes with one function. It takes the form

```
Function:       CHARCODE
# Arguments:    1
Argument:       1) an S-expression, EXPRESSION
Value:          An atom or list with all characters
                replaced by their corresponding character
                codes.
```

CHARCODE is an NLAMBDA function. Consider the following example:

```
←(CHARCODE x)
88
←(CHARCODE "D")
68
←(CHARCODE (M I C H E N E R))
(77 73 67 72 69 78 69 82)
```

CHARCODE is especially useful when you must specify non-printing ASCII characters. A control character may be represented by preceding a character with ⟨↑⟩. For example,

```
←(CHARCODE ⟨↑⟩B)
2
```

If an atom or string begins with #, CHARCODE interprets it as an indication of a *meta-character*. Normally, ASCII uses the integers 0 to 127 to represent characters. However, bytes are normally accorded a length of 8 bits, so there are another 128 integers (128–255) that are unused. By preceding a character with #, the character code that is returned is translated to the extended ASCII range. For example,

```
←(CHARCODE #X)
216
```

CHARCODE provides atoms for the most frequently used non-printing characters:

Character Code	Atom
13	CR
10	LF
32	SPACE, SP
27	ESC, ESCAPE
7	BELL
8	BS (e.g., backspace)
9	TAB
0	NULL
127	DEL

CHARCODE also maps NIL into NIL because some character manipulation functions can return NIL as their value.

9.4.4 Character Translation

One of the most difficult problems that many users face in transporting programs from one computer system to another is the translation from one character set to another. There are two major character sets: ASCII and EBCDIC. ASCII is used by INTERLISP-10, INTERLISP/VAX, and INTERLISP-D. EBCDIC is used by INTERLISP/370. This section describes a character translation function for converting from ASCII to EBCDIC (since it is the least frequent direction of conversion).

The translation procedure may be defined as a table lookup procedure. We will use the character code of the ASCII character as an index into the table to determine the corresponding EBCDIC value.

We may initialize the EBCDIC array as follows:

```
(DEFINEQ
    (initialize-ebcdic-array NIL
        (PROG (ebcdic-table)
```

```
(SETQ ebcdic-table (array 256 0 'FIXP)
(FOR I FROM 0 TO 75
     DO
         (SETA ebcdic-table I I))
(SETA ebcdic-table 76 (CHCON1 '<))
(SETA ebcdic-table 77 (CHCON1 '[))
(SETA ebcdic-table 78 (CHCON1 '+))
(SETA ebcdic-table 79 (CHCON1 '))
(SETA ebcdic-table 80 (CHCON1 '&))
(FOR I FROM 8 TO 89
     DO
         (SETA ebcdic-table I I))
(SETA ebcdic-table 90 (CHCON1 '!))
(SETA ebcdic-table 91 (CHCON1 '$))
(SETA ebcdic-table 92 (CHCON1 '*))
(SETA ebcdic-table 93 (CHCON1 ')))
(SETA ebcdic-table 94 (CHCON1 ';))
(FOR I FROM 95 TO 108
     DO
         (SETA ebcdic-table I I))
(SETA ebcdic-table 109 (CHCON1 ' ))
(SETA ebcdic-table 110 (CHCON1 '>))
(SETA ebcdic-table 111 (CHCON1 '?))
(FOR I FROM 112 TO 120
     DO
         (SETA ebcdic-table I I))
(SETA ebcdic-table 121 (CHCON1 ''))
(SETA ebcdic-table 122 (CHCON1 ':))
(SETA ebcdic-table 123 (CHCON1 '#))
(SETA ebcdic-table 124 (CHCON1 '@))
(SETA ebcdic-table 125 (CHCON1 ''))
(SETA ebcdic-table 126 (CHCON1 '=))
(SETA ebcdic-table 127 (CHCON1 '"))
(SETA ebcdic-table 128 128)
(FOR I FROM 129 TO 137
     DO
         (SETA ebcdic-table
               I
               (CHCON1
                   (L-CASE
                        (CHARACTER I)
                        NIL))))
```

Now, we have to be able to handle conversion of several different types of data objects. Our function to perform this translation for atoms is given below:

```
(DEFINEQ
   (convert.to.ebcdic (x)
      (PROG (result)
            (COND
               ((ATOM x)
                  (RETURN
                     (PACK
                        (MAPCONC (UNPACK x)
                                 (FUNCTION lookup)))))))
   ))
```

As an exercise, you might consider how to include strings and lists as datatypes to be handled by the COND clause.

9.5 DETERMINING PNAME LENGTH

When constructing a formatted buffer for printing, we often need to know the number of characters comprising the PNAME of an atom or string in order to avoid overflowing the buffer. INTERLISP provides **NCHARS** to tell us how many characters make up the PNAME of its argument. For example,

```
←(NCHARS 'hydrogen)
8
←(NCHARS "einsteinium" T)
13
```

because the "s are included as part of the PNAME.

```
←(NCHARS 1756.7)
6
←(NCHARS "Now is the time for all good men")
34
```

The generic format for calling NCHARS is

```
Function:       NCHARS
# Arguments:    1-3
Arguments:      1) an atom or string, ATM
                2) a flag, FLAG
                3) a read table, RDTBL
Value:          The number of characters comprising the
                PNAME or NIL.
```

If the value of FLAG is T, then NCHARS uses the PRIN2-PNAME of its first argument. If RDTBL is non-nil (see Section 14.4), it is used to interpret the characters comprising the PRIN2-PNAME of its first argument.

We can define a basic form of NCHARS for PRIN1-PNAMEs as follows:

```
(DEFINEQ
     (nchars (argument)
          (COND
               ((OR
                    (ATOM argument)
                    (STRINGP argument)
                    (NUMBERP argument))
                    (LENGTH (UNPACK argument))))
     ))
```

9.6 EXTRACTING CHARACTERS

You may want to extract the Nth character of the PNAME of an atom. **NTHCHAR** takes the form

```
Function:       NTHCHAR
                NTHCHARCODE

# Arguments:    4

Arguments:      1) an atom or string, ATM
                2) an index, N
                3) a flag, FLAG
                4) a readtable, RDTBL

Value:          The Nth character of ATM otherwise, NIL.
```

NTHCHAR returns the Nth character of the PNAME of ATM. N may be positive, whence the character is extracted relative to the beginning of the name, or negative, whence it is extracted relative to the end of the name. NTHCHAR returns NIL if N is greater than (NCHARS x) or less than (MINUS (NCHARS x)). For example,

```
←(NTHCHAR 'BALTIMORE 5)
I

←(NTHCHAR 'BALTIMORE)
NON-NUMERIC ARG
NIL
```

because we have not specified any value for N.

```
←(NTHCHAR "BALTIMORE" 5)
I
```

An alternative form, **NTHCHARCODE**, returns the character code of the Nth character. For example,

```
←(NTHCHARCODE 'BALTIMORE 5)
73
←(NTHCHARCODE 'BALTIMORE 0)
NIL
←(NTHCHARCODE 1.345 3)
51
←(NTHCHARCODE 'BALTIMORE -4)
77
```

If FLAG is T, either function uses the PRIN2-PNAME of ATM mediated by the read table (see Section 14.4).

A Definition for NTHCHAR

We might define NTHCHAR as follows:

```
(DEFINEQ
     (nthchar (atm n)
          (COND
               ((OR
                    (ATOM atm)
                    (STRINGP atm)))
               (T
                    (ERROR "ILLEGAL ARG" atm)))
          (COND
               ((GREATERP (ABS n)
                          (LENGTH (UNPACK atm)))
                    NIL)
               ((GREATERP n 0)
                    (CADR (LASTN (UNPACK atm) n)))
               ((LESSP n 0)
                    (CADR (LASTN (UNPACK atm)
                                 (PLUS (LENGTH (UNPACK
                                 atm))
                                       n)))))
     ))
```

9.7 SELECTING ALTERNATIVES BY CHARACTER CODES

Many applications use single characters as commands. **SELCHARQ** allows you to branch to different alternatives based on the value of a single character code. This function is also heavily used in writing communications software to assist in the deciphering of protocol characters generated by different host systems. It takes the form

```
Function:       SELCHARQ
# Arguments:    2-N
Arguments:      1) an expression, EXPRESSION
                2-N) clauses, CLAUSE[i]
Value:          The value of the last expression in the
                selected set of clauses.
```

SELCHARQ operates like SELECTQ (see Section 3.6). However, it uses the character code equivalent of the value of EXPRESSION rather than the quoted character itself. Each clause takes the form

```
(⟨key⟩ ⟨action⟩)
```

where ⟨key⟩ is a single character or a list of characters to be matched against the value of EXPRESSION. When a match is found, all of the expressions appearing in ⟨action⟩ are executed. The value of SELCHARQ is the value of the last expression executed in the clause. Matching is performed using EQ for single characters or MEMB for a list of characters. There must be a default clause which is the last clause in the list of selectors. If no match is found, the expressions in the default clause are executed. The value of SELCHARQ is then the value of the last expression executed in the default clause.

SELCHARQ is an NLAMBDA, nospread function.

9.8 CASE FUNCTIONS

INTERLISP normally operates in upper case. That is, it accepts all commands and expressions in upper case. However, it makes provision for accepting lower case characters in certain instances. The body of a comment may be lower case as may the contents of a string. When matching strings, you must be careful to ensure that they correspond in case at every character position. Certain CLISP words may also be entered in lower case, whence CLISP performs the appopriate translation before executing the statement.

INTERLISP provides several functions for translating from one case to the other and for testing the case of an object.

L-CASE translates an object to lower case while **U-CASE** translates an object to upper case. They take the form

```
Function:       L-CASE

# Arguments:    2

Arguments:      1) an object, X
                2) a flag, FLG

Value:          The lower case representation of X.
```

L-CASE produces a lower case version of X. If FLG is T, the first letter will be capitalized. For example,

```
←(L-CASE 'WICHITA)
wichita

←(L-CASE 'WICHITA T)
Wichita

←(L-CASE "FILE NOT FOUND")
"file not found"
```

If X is a list, L-CASE returns a new list with L-CASE applied to each element of the list. For example,

```
←(L-CASE '(MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY) T)
(Monday Tuesday Wednesday Thursday Friday)
```

U-CASE takes the form

```
Function:       U-CASE
                U-CASEP

# Arguments:    1

Argument:       1) an object, X

Value:          An upper case representation of X.
```

Consider the following example:

```
←(U-CASE "As I was walking to St. Ives")
"AS I WAS WALKING TO ST. IVES"
```

You may test if an object is an upper case representation using the predicate **U-CASEP**. U-CASEP returns T if its argument, an object X, contains no lower case characters. Consider the following example:

```
←(U-CASEP '(SATURDAY SUNDAY))
T
```

10

String Manipulation Functions

A string is a sequence of zero or more alphanumeric and/or special characters that represents a literal value. Unlike atoms, strings do not represent memory locations within INTERLISP and, therefore, do not have values except the string representation itself. A string is demarcated by " (double quote). A string may be assigned as the value of an atom.

INTERLISP provides a comprehensive set of functions for creating and manipulating strings. This chapter describes these functions and some applications demonstrating how strings may be used.

10.1 CREATING A STRING

The basic function for creating a string is **MKSTRING**, which takes the following format

Function:	MKSTRING
# Arguments:	1
Argument:	1) an atom or list, X
Value:	A string corresponding to the PRIN1-PNAME of X.

Consider the following examples:

```
←(MKSTRING)
"NIL"
←(MKSTRING 'x)
"X"
```

where the PRIN1-PNAME of 'X is X.

When a string is created, INTERLISP builds an internal data structure consisting of a string pointer and the sequence of characters that comprise the string. The string pointer contains the storage location where the sequence of characters begins and the number of characters comprising the string. Several string pointers may reference the same set of characters. String pointers may also point into the middle of a sequence of characters (as a result of CONCAT or SUBSTRING). This approach guarantees efficient management of string storage space.

We may also create a string by assigning it as a value to an atom. For example,

```
←(SETQ a-string
       "Go therefore and make disciples of all nations")
"Go therefore and make disciples of all nations"
```

implicitly performs an MKSTRING to create the string data structure. The value of the atom is the string pointer.

If we give MKSTRING a list as its argument, it makes a string of the whole list (including the parentheses). For example,

```
←(SETQ proverb
       (LIST 'a 'stitch 'in 'time 'saves 'nine))
(a stitch in time save nine)
←(SETQ cliche (MKSTRING proverb))
"(a stitch in time saves nine"
```

But, this is not what we want! The result includes the parentheses. To remedy the situation, let us write a recursive procedure that converts a list into a string:

```
(DEFINEQ
     (make.string.from.list (lst)
          (CONCAT (MKSTRING (CAR lst)
                  (COND
                     ((NULL lst) (MKSTRING))
                     ((make.string.from.list (CDR lst))))
     ))
```

Now, let us apply this function to the previous example:

```
←(SETQ cliche (MAKE.STRING.FROM.LIST proverb))
"a stitch in time saves nine"
```

10.1.1 Allocating a String Pointer

You may create a string pointer of a given length and initialize it to a default character value using **ALLOCSTRING**. It takes the form

```
Function:       ALLOCSTRING
# Arguments:    3
Arguments:      1) the length of the string, N
                2) the initialization character, INITCHAR
                3) an old string pointer, OLDPTR
Value:          A string of length N initialized to
                INITCHAR.
```

INITCHAR must be a character code or an expression that is coercible to a character code. If INITCHAR is an atom, the first character of the atom is used as the initialization character. If INITCHAR is NIL, it defaults to the character code N (from NIL). Consider the following examples:

```
←(ALLOCSTRING 1)
"N"

←(ALLOCSTRING 3 0)
"""

←(ALLOCSTRING 5 'B)
"BBBBB"

←(ALLOCSTRING 10 'HELP)
"HHHHHHHHHH"

←(ALLOCSTRING 4 (LIST 'a 'b))
"((((""

←(ALLOCSTRING)
NON-NUMERIC ARG
NIL
```

because a string must have a length greater than 0.

10.2 EXTRACTING SUBSTRINGS

Strings, from an external viewpoint, are indivisible objects in INTERLISP. That is, each function manipulates the entire contents of the string. However, we often need to break strings down into their constituent parts (for example, when processing textual data). Two major operations are necessary:

1. Extracting a substring
2. Peeling characters from the beginning or end of a string one-by-one

10.2.1 The SUBSTRING Function

Given a string, we can create a new string by extracting a portion of it—the *substring* operation. The function **SUBSTRING** allows us to extract pieces of a string. The format for SUBSTRING is

```
Function:       SUBSTRING
# Arguments:    4
Arguments:      1) an S-expression whose value is a
                   string, STRING
                2) the index of the first character of the
                   intended substring, N
                3) the index of the last character of the
                   intended substring, M
                4) an optional string pointer, OLDPTR
Value:          The specified substring; otherwise, NIL.
```

SUBSTRING extracts the Nth through Mth characters of the string. If M is NIL, SUBSTRING extracts the Nth through last characters of the string. N may not be NIL. Both N and M may be negative numbers, thus referring to the end of the string rather than its beginning.

```
←(SETQ STR "WELCOME TO THE THEATRE")
"WELCOME TO THE THEATRE"

←(SUBSTRING STR 9 14)
"TO THE"

←(SUBSTRING STR 16)
"THEATRE"

←(SUBSTRING STR NIL 7)
NON-NUMERIC ARG
NIL

←(SUBSTRING STR 14 9)
NIL

←(SUBSTRING STR -14 -9)
"TO THE"
```

If the string is not defined, e.g., then SUBSTRING applies MKSTRING to the first argument before extracting the subtring.

N and M must be well-defined according to the following conditions:

N ⟨ M

N ⟨ (NCHARS string) and M ⟨ (NCHARS string), if M not NIL

If N or M is negative, then N (or M) ⟨ (MINUS (NCHARS string))

If OLDPTR is a string pointer, it is reused to perform the subtring operation; otherwise, a new string pointer will be created.

10.2.2 Getting the Next or Last Character

Many parsing programs, particularly command recognizers, need to extract characters from a string one-by-one. **GNC** allows you to get the next character from a string. GNC maintains an internal pointer to the string. It does not physically remove characters from the string, but merely changes the pointer and the byte count. Thus, when you print the string, it appears as if characters have been removed because the pointer has been changed.

If the argument is not a string, GNC applies MKSTRING to it before extracting the specified character. If the argument is NIL or the null string, GNC returns NIL.

GLC allows you to get the last character of a string. As with GNC, the pointer and the byte count are changed. Together, these two functions allow you extract characters from either end of the string.

These functions take the following format

```
Function:        GNC
                 GLC

# Arguments:     1

Argument:        1) a string, X

Value:           The next or last character of X.
```

Consider the following example:

```
←(SETQ z "Now is the time for all good men")
"Now is the time for all good men"

←(for I from 1 to 32 do (PRIN1 (GNC z))(SPACES 1))
N o w i s t h e t i m e f o r a l l g o o d m e n

←(for I from 1 to 32 do (PRIN1 (GLC z))(SPACES 1))
n e m d o o g l l a r o f e m i t e h t s i w o N

←(GNC)
N
```

because it applies (MKSTRING) which yields "NIL" and

```
←(GLC)
L
```

for the same reason.

```
←(SETQ string "rumpelstiltskin")
"rumpelstiltskin"
←(GNC string)
R
←string
"umpelstiltskin"
←(GLC string)
N
←string
"umpelstiltski"
```

Thus, if you need to preserve the original value of the argument, you should copy it before applying either GNC or GLC to it. Consider the following example:

```
←(SETQ string "abc")
"abc"
←(GNC string)
a
←(GNC string)
b
←(GNC string)
c
←(GNC string)
NIL
```

because the value of STRING is now the null string.

10.3 CONCATENATING STRINGS

The functions in Section 10.2 allow you to take strings apart. **CONCAT** allows you to put two or more strings together into a new string. The format for CONCAT is

```
Function:       CONCAT
# Arguments:    1-N
```

```
Arguments:      1) a string, STRING[1]
                2-N) strings, STRING[2] ... STRING[n]
Value:          A new string composed of the individual
                arguments.
```

CONCAT is a nospread function. CONCAT creates a new string pointer and copies each of the argument strings to the new string. If any of the arguments are not strings, CONCAT applies MKSTRING to it before copying. Consider the following example:

```
←(CONCAT)
""                      e.g., the null string.
←(CONCAT "Baltimore" " " 'is " " 'best.)
"Baltimore is best."
```

10.3.1 Concatenating a List of Objects

CONCATLIST concatenates a list of strings or other INTERLISP objects to form a string. It takes the form

```
Function:       CONCATLIST
# Arguments:    1
Argument:       1) a list of objects, LST
Value:          A new string that is the concatenation of
                the individual elements of LST.
```

If the elments of LST are not strings, MKSTRING is applied to each in turn to produce a string which is concatenated into the result. Consider the following examples:

```
←(CONCATLIST)
""
```

which is the null string.

```
←(CONCATLIST (LIST 'a 'b 'c 'd 'e 'f))
''ABCDEF"
```

A Definition for CONCATLIST
We might define CONCATLIST as follows:

```
(DEFINEQ
    (concatlist (lst)
```

```
        (APPLY (FUNCTION CONCAT) lst)
    ))
```

10.4 TESTING STRINGS

There are two types of tests that we would like to apply to strings. The first determines if the argument is a string. The second determines if two strings are equal.

10.4.1 Determining String Existence

STRINGP allows you to determine whether a given S-expression is a string or not. It returns the value of the S-expression if it is a string; otherwise, NIL. It takes the form

```
Function:       STRINGP
# Arguments:    1
Argument:       1) a string, STRING
Value:          The value of STRING if it is a string;
                otherwise, NIL.
```

Consider the following examples:

```
←(STRINGP "Baltimore is best.")
"Baltimore is best."
←(STRINGP)
NIL
←(STRINGP 1.56)
NIL
```

10.4.2 Testing the Equality of Strings

STREQUAL allows you to determine if two strings, X and Y, are equal. Equality is decided by determining whether or not the strings will print the same. Strings may be equal without satisfying EQ as explained in Section 4.6. STREQUAL takes the form

```
Function:       STREQUAL
# Arguments:    2
Arguments:      1) a string, X
                2) a string, Y
Value:          T, if the strings are equal.
```

Consider the following examples:

```
←(STREQUAL "New York" "New York")
T
←(EQ "Washington" "Washington")
NIL
←(EQUAL "Washington" "Washington")
T
```

Two separate string pointers are created by INTERLISP when strings are read in from the terminal (by RSTRING—see Section 14.2.4). Thus, the two string pointers in the example above are not EQ, although their contents are EQUAL. Note that EQUAL uses STREQUAL to determine the equality of two strings.

10.4.3 Testing String Membership

In many cases, we want to know if one string exists within another string. **STRMEMB** determines if its first argument, a string, is contained within its second argument, another string. It takes the form

```
Function:       STRMEMB
# Arguments:    2
Arguments:      1) a string, X
                2) a string, Y
Value:          The substring of Y, if X is contained
                within Y; otherwise, NIL.
```

Consider the following examples:

```
←(STRMEMB "X" "TAX YEAR")
"X YEAR"
```

STRMEMB returns the substring of Y beginning with X if X is included within Y.

A Definition for STRMEMB
We might define STRMEMB as follows:

```
(DEFINEQ
     (strmemb (x y)
          (PROG (achar index)
                (SETQ y (SUBSTRING y 1))
```

```
        loop1
              (SETQ index 1)
        loop2
              (SETQ achar (NTHCHAR x index))
              (COND
                  ((NULL achar)
                       (RETURN y)))
              (COND
                  ((EQ achar (NTHCHAR y index)
                       (SETQ index (ADD1 index))
                       (GO loop2)))
              (COND
                  ((NULL (GNC y))
                       (RETURN NIL))
                  (T   (GO loop1))))
))
```

10.5 REPLACING ELEMENTS OF A STRING

In many programs, as we process strings, we want to replace characters in the string by new characters. **RPLSTRING** allows you to substitute characters within strings. The format of RPLSTRING is

```
Function:       RPLSTRING
# Arguments:    3
Arguments:      1) an original string, X
                2) the index of substitution, N
                3) the substitution string, Y
Value:          The modified version of the string X.
```

RPLSTRING replaces characters in X beginning at the Nth character with characters from Y. Substitution continues until either Y is exhausted or the length of X is exceeded. Replacement is a one-for-one substitution of characters in X. If there is not enough room in X to accommodate the new string, an error occurs. X is physically modified by this operation. Note that if X is a substring of some other string, say Z, then Z is modified also. Thus, you should exercise caution concerning the indiscriminate modification of strings as a ripple effect may occur that was not intended.

N may be positive or negative, but may not be greater than (NCHARS x). If either X or Y is not a string, it is converted to a string before the replacement operation is executed. In this case, a new string pointer will be returned,

```
←(SETQ x "ABCDEF")
"ABCDEF"
```

```
←(RPLSTRING x -2 "XYZ")
ILLEGAL ARG
XYZ
```

because inserting "XYZ" into X would increase its length, which is not allowed.

```
←(RPLSTRING x 2 "XYZ")
"AXYZEF"
```

Note that "BCD" have been replaced by "XYZ".

```
←(SETQ x "BALITMORE")
"BALITMORE"
```

which is misspelled!

```
←(SETQ y (SUBSTRING x 3 6))
"LITM"
←(RPLSTRING y 2 "TI")
"LTIM"
←x
"BALTIMORE"
```

where X is modified because Y was a substring of X.

To insert without modifying the string, see INSERT.STRING in Section 10.7.

10.5.1 Replacing Elements with Character Codes

An alternative form of RPLSTRING is RPLCHARCODE. **RPLCHARCODE** is used primarily to insert nonprinting character codes into strings. It takes the form

```
Function:       RPLCHARCODE
# Arguments:    3
Arguments:      1) an original string, X
                2) an index, N
                3) a character code, CHARCODE
Value:          A modified version of the string X.
```

RPLCHARCODE is used to replace (or place) a single element of a string with a specified character code. The index N may be positive or negative. If N is negative, replacement is relative to the end of the string.

10.6 SEARCHING A STRING

One of the most common string operations is to search one string looking for an occurrence of another string. **STRPOS**, which implements the string searching operation, takes the form

```
Function:       STRPOS
# Arguments:    6
Arguments:      1) a pattern string, PATTERN
                2) a string, STRING
                3) a starting position, START
                4) a skip (wild card) character, SKIP
                5) an anchor flag, ANCHOR
                6) a tail flag, TAIL
Value:          The character position in STRING if the
                match is successful.
```

If either PATTERN or STRING is not a string, it is automatically converted (using MKSTRING) before searching begins. The search starts at the character indexed by START. If START is NIL, 1 is assumed. STRPOS looks for a sequence of characters in STRING that match PATTERN. If a match is found, the character index of the first matching character of the sequence of characters is returned as the value of STRPOS. Otherwise, NIL is returned. Consider the following examples:

```
←(SETQ x "the quick brown fox jumped over the lazy dog")
"the quick brown fox jumped over the lazy dog"
←(STRPOS "fox" x)
17
←(STRPOS "fox" x 10)
17
←(STRPOS "fox" x 20)
NIL
←(STRPOS "fox" X NIL)
17
←(STRPOS "fox" x -30)
17
←(STRPOS "" "")
NIL
←(STRPOS)
1
```

Searching may be modified in several ways:

1. SKIP is a character that acts like a wild card; that is, wherever it occurs in PATTERN it matches any character in string in the corresponding position. SKIP may be any character but it is best to pick one that is not likely to occur in the string that you are searching. For example,

```
←(STRPOS "br&wn" x NIL '&)
11
←(STRPOS "dog&" x 1 '&)
NIL
```

2. If ANCHOR is T, STRPOS searches for a match only at START (or 1, if START is NIL). If a match fails between PATTERN and the character sequence of STRING beginning at START, STRPOS returns NIL. For example,

```
←(STRPOS "fox" x)
17
←(STRPOS "fox" x 17 NIL T)
17
←(STRPOS "fox" x 15 NIL T)
NIL
```

3. If TAIL is T, the character index returned by STRPOS is the index of the first character after the PATTERN was found in STRING, i.e., the tail string. For example,

```
←(STRPOS "fox" x 1 NIL NIL T)
20
←(STRPOS "fox" x 17 NIL T T)
20
←(STRPOS "fox" x 15 NIL T T)
NIL
```

STRPOS may return a character position outside the string. For example,

```
←(STRPOS "dog" x 1 NIL NIL T)
45
```

even though the string is only 44 characters long. Care should be taken in using this feature of STRPOS in conjunction with RPLSTRING because of the possibility of errors that may be generated by RPLSTRING.

A Definition of STRPOS

We might define STRPOS as follows:

```
(DEFINEQ
    (strpos (pattern string start skip anchor tail)
        (*
            Make PATTERN a string no matter what it is
            originally!
        )
        (COND
            ((STRINGP pattern))
            ((LITATOM pattern)
                (SETQ pattern (MKSTRING pattern)))
            ((NULL (STRINGP pattern))
                (SETQ pattern (MKSTRING pattern))))
        (*
            Make STRING a string no matter what
            datatype it is originally!
        )
        (COND
            ((STRINGP string))
            ((LITATOM string)
                (SETQ string (MKSTRING string)))
            (T
                (SETQ string (MKSTRING string))))
        (*
            The SKIP character must be a single
            character which the following code
            assures, no matter how many characters are
            provided.
        )
        (COND
            (skip
                (SETQ skip (NTHCHAR skip 1))))
        (*
            Orient START, if it is defined, to the
            beginning of the string.
        )
        (COND
            (start
                (COND
                    ((MINUSP start)
                        (SETQ start
                            (IPLUS start
```

```
                                    (NCHARS
                                    string)
                                    1)))))
        (T
            (SETQ.start 1)))
(*
    Now, isolate the proper substring to be
    searched rather than searching from the
    beginning of the string each time.
)
(SETQ string
    (SUBSTRING string start))
(*
    Search for PATTERN in STRING
)
(PROG (achar substring.x substring.y index)
    (SETQ index start)
loop2
    (*
        Get the first character of the
        respective strings.
    )
    (SETQ substring.x (SUBSTRING pattern 1))
    (SETQ substring.y (SUBSTRING string 1))
loop1
    (COND
        ((SETQ achar (GNC substring.x))
            (COND
                ((EQ achar (GNC
                substring.y))
                    (GO loop1))
                ((EQ achar skip)
                    (GO loop1))
                (T
                    (GO next.character))))
        (tail
            (RETURN
                (IPLUS (NCHARS pattern)
                    index)))
        (T
            (RETURN index)))
next.character
    (*
        If no match in the exact position at
        START, cease further searching.
    )
```

```
                (COND
                    (anchor
                        (RETURN)))
                (*
                    Strip a character from STRING and
                    proceed if any characters left to
                    search.
                )
                (COND
                    ((GNC string)
                        (SETQ index (ADD1 index))
                        (GO loop2))
                    (T
                        (RETURN))))
))
```

10.6.1 Searching a String for a Character

In many cases, you will want to search a string for the first occurrence of a character which may be a member of a set of characters. **STRPOSL**, which compares a set of characters against a string until one matches, takes the form

```
Function:       STRPOSL
# Arguments:    4
Arguments:      1) a list of characters, CHARSET
                2) a string to be searched, STRING
                3) A starting index, START
                4) a non-membership flag, NEG
Value:          A character index or NIL.
```

CHARSET may be a list of characters or character codes. STRPOSL searches STRING beginning at START (or 1, if START is NIL) for one of the characters in CHARSET. If one character matches, the character index of the matching character is returned. If no match occurs, NIL is returned. Consider the following example:

```
←(STRPOSL '(Q Z J) x)
5
```

where X is defined as above.

If NEG is T, then STRPOSL finds the first character which is *not* a member of ASET.

```
←(STRPOSL '(A B C D E F G H I K L M N O P Q R S T U V X
          Y Z) x 1 T)
4
```

because SPACE is a legal character. If you insert a space at the end of the first argument (using the form %⟨space⟩), re-executing the expression yields 14, which is the index of W which does not appear in CHARSET.

A Definition of STRPOSL

We may define a simple form of STRPOSL as follows:

```
(DEFINEQ
     (strpos1 (charset string)
          (PROG (index)
                (SETQ string (MKSTRING string))
                (SOME charset
                      (FUNCTION strpos1))
                (RETURN index))
     ))
```

where STRPOS1 is defined as

```
(DEFINEQ
     (strpos1 (achar)
          (SETQ index (STRPOS achar string))
     ))
```

10.6.2 Creating Bit Tables

String searching is enhanced by converting the character codes to bit representations. STRPOSL will automatically convert the characters (or their codes) in CHARSET into a bit representation if CHARSET is not a bittable. To do so, it uses the function **MAKEBITTABLE**, which takes the form

```
Function:       MAKEBITTABLE

# Arguments:    3

Arguments:      1) a list of character codes, CHARSET
                2) a non-membership flag, NEG
                3) an array, A

Value:          An array with bit representations for the
                characters in LST.
```

CHARSET is a list of characters or character codes as specified for STRPOSL. NEG is the same as used by STRPOSL. MAKEBITTABLE returns a bit table as an array containing the bit representations of the characters in CHARSET. For example,

```
←(SETQ charset '(A B C D E F G H I J K))
(A B C D E F G H I J K)

←(SETQ y (MAKEBITTABLE charset))
{ARRAYP}#542635
```

If A is an array, it is modified and returned as the new bittable to be used by STRPOSL.

10.7 STRING OPERATIONS

Given a string, we often want to insert, delete, or substitute for elements of the string. RPLSTRING can be coded to provide these functions, but it smashes the characters into the string given as its argument. The following functions provide similar capabilities, but return a new string composed of the appropriate elements of the old. These functions are not part of standard INTERLISP at this time.

These functions have been defined very simply. No doubt you can increase their complexity with a little extra thought. They are merely intended to show that you do not have to rely entirely on RPLSTRING for manipulating strings.

10.7.1 Inserting into a String

Given a string of arbitrary length, we often want to insert a new string into the middle of the string. **INSERT.STRING** takes the form

```
Function:       INSERT.STRING

# Arguments:    3

Arguments:      1) a string, X
                2) a fragment to be inserted, FRAGMENT
                3) a position, POS

Value:          A new string.
```

INSERT.STRING inserts the fragment immediately after POS. Consider the example

```
←(INSERT.STRING "XYZ" "ABC" 1)
"XABCYZ"
```

A Definition for INSERT.STRING
We might define INSERT.STRING as follows:

```
(DEFINEQ
    (insert.string (x fragment pos)
        (COND
            ((NOT (STRINGP x)) NIL))
        (COND
            ((NOT (STRINGP fragment))
                (SETQ fragment (MKSTRING fragment))))
        (CONCAT
            (SUBSTRING x 1 pos)
            fragment
            (SUBSTRING x (add1 pos)))
    ))
```

10.7.2 Deleting from a String

DELETE.STRING deletes a substring from a string and returns a new string which is the concatenation of the remaining parts. It takes the form

```
Function:       DELETE.STRING

# Arguments:    3

Arguments:      1) a string, OLD
                2) a starting position, N
                3) an ending position, M

Value:          A new string with the appropriate
                characters deleted.
```

Consider the following example:

```
←(DELETE.STRING "ABCDEFG" 3 5)
"ABFG"
```

A Definition for DELETE.STRING
We might define DELETE.STRING as follows:

```
(DEFINEQ
    (delete.string (old n m)
        (CONCAT
            (SUBSTRING old 1 (SUB1 n))
            (SUBSTRING old (ADD1 m)))
    ))
```

10.7.3 Substituting into a String

SUBSTITUTE.STRING substitutes a fragment in place of a portion of a string. It takes the form

```
Function:       SUBSTITUTE.STRING

# Arguments:    4

Arguments:      1) a string, OLD
                2) a string to substitute, FRAGMENT
                3) a starting position, N
                4) a final position, M

Value:          A new string with FRAGMENT substituted for
                the substring identified by (N,M).
```

Consider the following example:

```
←(SETQ string "ABCDEFG")
"ABCDEFG"

←(SUBSTITUTE.STRING string "XYZ" 3 5)
"ABXYZFG"
```

A Definition for SUBSTITUTE.STRING
We might define SUBSTITUTE.STRING as follows:

```
(DEFINEQ
    (substitute.string (old fragment n m)
        (CONCAT
            (SUBSTRING old 1 (SUB1 n))
            fragment
            (SUBSTRING old (ADD1 m)))
    ))
```

10.8 TRIMMING A STRING

Many strings, particularly those created by reading or processing text files, contain an excess of blanks. Usually, we would like to eliminate these blanks in order to tidy up the appearance of the string. **TRIM** will remove the excess blanks from a string. It takes the following form

```
Function:       TRIM

# Arguments:    1

Argument:       1) a string, STRING
```

```
Value:          A new string corresponding to STRING with
                all excess blanks removed.
```

TRIM returns NIL if its argument is not a string.

A Definition for TRIM

We might define TRIM as follows:

```
(DEFINEQ
    (trim (astring)
        (COND
            ((NOT (STRINGP astring)) NIL)))
        (PROG (newstring bflag achar)
            (*
                Create an empty string in which to
                compose the return value.
            )
            (SETQ newstring (MKSTRING))
        loop
            (SETQ achar (GNC astring))
            (COND
                ((NULL achar)
                    (*
                        GNC returns NIL when no
                        more characters are left to
                        process.
                    )
                    (RETURN newstring))
                ((EQUAL (CHCON1 achar) (CHARCODE " "))
                    (COND
                        (bflag
                            (*
                                BFLAG set if we
                                have seen one
                                blank already.
                            )
                            (GO loop))
                        (T
                            (*
                                Have not yet seen
                                a blank. Set
                                BFLAG to one and
                                keep this blank.
                            )
                            (SETQ bflag T)))
```

```
                    (T
                        (*
                            Some character other than a
                            blank.
                        )
                        (SETQ bflag NIL)))
                (SETQ newstring (CONCAT newstring achar))
                (GO loop))
))
```

Note: TRIM is not a standard INTERLISP function.

11

Array Manipulation Functions

Arrays are data structures that were added after the initial definition of LISP was developed. INTERLISP supports two types of arrays: basic linear arrays and hash tables. A linear array is a sequence of locations that contain values which are indexed by numerical position. A *hash array* is an array that relates two sets of pointers. The first pointer is known as the *hash item* while the second is called the *hash value*. Hash arrays operate in a manner similar to property lists in that they establish an association between the hash item and the hash value.

In Section 11.4, we will show you how to define and manage matrices, which are two-dimensional arrays. INTERLISP does not provide support for this datatype. But, with a few functions, you can develop a comprehensive matrix handling package that operates like any other datatype.

11.1 CREATING AN ARRAY: INTERLISP-10

An array may only be created in INTERLISP through a function invocation. To create an array, you use **ARRAY**, which takes the following format

```
Function:       ARRAY
# Arguments:    3
Arguments:      1) Size of the array, N
                2) Number of value cells, P
                3) Number of pointer cells, V
Value:          A pointer to the array printed as
                {ARRAYP}#⟨address⟩.
```

When you execute (ARRAY n p v), INTERLISP allocates a block of storage of size N+2 words. The first two words contain descriptive information about the array that is used internally by INTERLISP. The next P ⟨ = N cells contain

numeric values. Initially, the numeric cells have the value 0 when the array is created. Finally, the last N—P => 0 cells may contain pointers to list cells where both the CAR and CDR portions are available for storing information. They are initialized to the value of V.

```
←(SETQ A1 (ARRAY 10 10 0))
{ARRAYP}#542224

←(SETA A1 1 (CONS 'X 'Y))
NON-NUMERIC ARG
(X . Y)
```

because only numeric information may be stored in the numeric ("unboxed") region of an INTERLISP-10 array.

If P is NIL, INTERLISP assumes a value of 0 and creates an array of pointers.

```
←(SETQ A2 (ARRAY 10 0 0))
{ARRAYP}#542240

←(SETA A2 1 (CONS 'X 'Y))
(X . Y)

←(ELT A2 1)
(X . Y)

←(SETA A2 1 (LIST 'A 'B 'C))
(A B C)

←(SETD A2 1 (LIST 'X 'Y 'Z))
(X Y Z)

←(ELT A2 1)
(A B C)

←(ELTD A2 1)
(X Y Z)
```

In general, INTERLISP allocates storage for arrays from a common array space. If sufficient space does not exist for the array to be created, INTERLISP will attempt a garbage collection to gather space. If enough space is still not available, INTERLISP generates an error with the message "ARRAYS FULL".

The array facility provided by INTERLISP-10, INTERLISP/370, and INTERLISP-VAX is more primitive than that provided by INTERLISP-D (as discussed below). The number region of an INTERLISP-10 array may only store numbers, not pointers. These cells are not inspected during a garbage collection. All arrays in INTERLISP-10/370/VAX are indexed beginning with 1.

11.1.1 Creating an Array: INTERLISP-D

INTERLISP-D does not support the combined numeric value and pointer data structure. Rather, arrays have been extended so that you may specify the type of the value to be stored in the array. This approach provides you with more flexibility in using arrays to refer to collections of data.

The format for defining arrays in INTERLISP-D is as follows:

```
Function:       ARRAY

# Arguments:    4

Arguments:      1) Size of the array, SIZE
                2) Type of the array, TYPE
                3) Initialization value, INIT
                4) Origin of the array, ORIGIN

Value:          A pointer to the array printed as
                {ARRAY}#⟨address⟩
```

INTERLISP-D accepts a type from the following list: BIT, BYTE, WORD, FIXP, FLOATP, POINTER, or DOUBLEPOINTER. If the value of TYPE is NIL, INTERLISP-D creates an array with the default type of POINTER. If TYPE is a number (i.e., SIZE), INTERLISP-D defaults to an array of type FIXP (i.e., integers).

The initialization value INIT is used to set the value of each element of the array when it is created. If INIT is NIL, zero is assumed for all numeric type arrays and NIL for all other types.

Arrays in INTERLISP-D may have their origin indexed by either 0 or 1. If no origin is specified, the array will be indexed beginning with 1.

For example, we may create an array called NUMBERS by the following statement:

```
←(SETQ numbers (ARRAY 5 5))
{ARRAYP}#1,2150

←(SETQ more-numbers (ARRAY 5 'FIXP 4 0))
{ARRAYP}#1,2140

←(ELT more-numbers 0)
4
```

11.2 MANIPULATING ARRAYS

INTERLISP provides several functions for manipulating arrays. Basically, these functions provide the user with a foundation for creating a rich and complex environment for specific applications.

11.2.1 Obtaining the Array Size

You may obtain the size of an array by invoking the function **ARRAYSIZE**. It takes the form

```
Function:       ARRAYSIZE
# Arguments:    1
Arguments:      1) an array pointer, ARRAYPTR
Value:          The size of the array.
```

INTERLISP generates an error, ARG NOT ARRAY, if the argument is not an array object (i.e., one created by the ARRAY function). For example, using NUMBERS

```
←(ARRAYSIZE numbers)
5
```

11.2.2 Obtaining the Array Type

INTERLISP creates arrays that contain either numeric value cells, pointer cells, or a mixture of both (INTERLISP-10/VAX). To obtain the type of the array, you may invoke the function **ARRAYTYP** with an array object as its argument. If the argument is not an array object, INTERLISP generates an error, ARG NOT ARRAY. It takes the form

```
Function:       ARRAYTYP
# Arguments:    1
Argument:       1) an array address, XARRAY
Value:          The second argument to ARRAY.
```

ARRAYTYP returns the value of the second argument to ARRAY. If this value is a positive number greater than zero, it indicates the number of numeric values that may be stored in the array (INTERLISP-10/VAX). If this number is 0 or NIL, it indicates that the array is composed entirely of pointers (INTERLISP-10/VAX).

```
←(ARRAYTYP A2)
0
```

which shows that A2 is an array defined to have only pointers as values.

Thus, if you need to determine the number of pointers that may be stored in an array, you must compute this number by subtracting the value of ARRAYTYP from ARRAYSIZE. We can encapsulate this in a function as follows

```
(DEFINEQ
    (arrayptr (array-object)
        (IDIFFERENCE
                (ARRAYSIZE array-object)
                (ARRAYTYP array-object))
    ))
```

ARRAYTYP is not defined in INTERLISP/370.

INTERLISP-D Convention
INTERLISP-D will return a value (i.e, the name of the datatype) that may be given to ARRAY (see above) that will generate the same type of array again. For example, using NUMBERS

```
←(ARRAYTYP numbers)
FIXP
```

11.2.3 Validating an Array Pointer

You may determine whether or not a pointer points to an array element by invoking the function **ARRAYP**. It takes the form

Function:	ARRAYP
# Arguments:	1
Arguments:	1) an array pointer, ARRAYPTR
Value:	The value of ARRAYPTR if it is an array pointer; otherwise, NIL.

ARRAYP is a predicate that returns the value of the pointer if it points to or into an array. Otherwise, it returns NIL. INTERLISP does not check to see whether or not the argument actually points to the beginning of the array. For example,

```
←(ARRAYP numbers)
{ARRAYP}#1,2150
```

11.2.4 Obtaining a Pointer to the Beginning of an Array

Given a pointer to an array element, you may obtain a pointer to the beginning of the array by invoking the function **ARRAYBEG**. ARRAYBEG takes as its argument a pointer into an array. It returns a pointer to the beginning of the array if the pointer is valid. Otherwise, it returns NIL.

INTERLISP-D does not support the function ARRAYBEG since it treats all arrays as linear objects of a given type. Rather, INTERLISP-D provides the function **ARRAYORIG** to return the origin of an array. It takes the form

```
Function:       ARRAYORIG
# Arguments:    1
Argument:       1) an array address, XARRAY
Value:          The origin of the array.
```

Consider the following examples:

```
←(ARRAYORIG numbers)          "INTERLISP-D"
1
←(ARRAYORIG A2)               "INTERLISP-10"
1
```

ARRAYORIG generates an error, ARG NOT ARRAY, if its argument does not satisfy ARRAYP.

In INTERLISP-10/VAX, ARRAYORIG always returns 1. ARRAYORIG is not defined in INTERLISP-370.

11.2.5 Setting the Value of an Array Element

You may set the value of the Ith array element by invoking the function **SETA**. It has the following format:

```
Function:       SETA
                SETD
# Arguments:    3
Arguments:      1) an array object, A
                2) an index, I
                3) a value, V
Value:          The value assigned to the Ith array
                element.
```

SETA sets the Ith element of the array A to the value V. If A is not an array object (i.e., returned by ARRAY), SETA generates an error ARG NOT ARRAY.

```
←(SETA a3 2 156)
156
```

```
←(ELT a3 2)
156
```

If I ⟨= P (from ARRAY), then the value of V must be numeric (i.e., satisfies NUMBERP). If P ⟨= I ⟨= N, V is assigned to the CAR of the Ith element. The latter case determines if the Ith element is in the pointer region of the array. Note that pointers may not be inserted into the numeric regions of INTERLISP-10/VAX arrays.

```
←(SETQ a3 (ARRAY 10 5 0))
{ARRAYP}#542270

←(SETA a3 2 (LIST 'ALEX 'ALICE 'ANDREA))
NON-NUMERIC ARG
(ALIX ALICE ANDREA)
```

because 2 indicates a cell in the numeric (unboxed) region of A3.

```
←(SETA a3 7 (LIST 'ALEX 'ALICE 'ANDREA))
(ALEX ALICE ANDREA)
```

because 7 indicates a cell in the pointer region of A3.

An alternative form of the function, **SETD**, will set the CDR of the Ith element of the array if I is an index within the pointer region of the array. For example,

```
←(SETD a3 7 (LIST 'BARRY 'BART 'BILL))
(BARRY BART BILL)

←(ELT a3 7)
(ALEX ALICE ANDREA)
```

because this list is stored in the CAR portion of the cell in the pointer region of A3 indicated by the index 7.

INTERLISP-D Conventions

Since INTERLISP-D supports typed arrays, I must always be less than or equal to N.

INTERLISP-D supports SETD only to maintain compatibility with INTERLISP-10. It treats SETD as SETA when SETD is invoked.

We can set the elements of the array NUMBERS using the following statements:

```
←(SETA numbers 1 1)
1
```

```
←(SETA numbers 2 2)
2
←(SETA numbers 3 3)
3
←(SETA numbers 4 4)
4
←(SETA numbers 5 5)
5
```

but when we attempt to set element 6:

```
←(SETA numbers 6 6)
ILLEGAL ARG
```

because the array NUMBERS is defined to be only five elements in length beginning with the element labeled 1.

11.2.6 Retrieving the Value of an Array Element

You may retrieve the value of the Ith array element by invoking the function **ELT**. It has the following format

Function:	ELT ELTD
# Arguments:	2
Arguments:	1) an array object, A 2) an index, I
Value:	The value of the Ith element of the array

If I ⟨= P (from ARRAY), ELT returns the numeric integer value contained in the Ith cell. If P ⟨ I ⟨= N, ELT returns the CAR of the Ith element of the array. In the latter case, I is an index into the pointer region of the array. For example,

```
←(ELT a3 7)
(ALEX ALICE ANDREA)
```

If A is not an array object, ELT generates an error ARG NOT ARRAY.

An alternative form, **ELTD**, returns the CDR of the Ith array element if I is an index into the pointer region of the array.

```
←(ELTD a3 7)
(BARRY BART BILL)
```

INTERLISP-D Conventions

Since INTERLISP-D supports typed arrays, the value of I must always be less than or equal to N.

INTERLISP-D supports ELTD to maintain compatibility with INTERLISP-10. It returns the same value as ELT. For example, to retrieve element 3 from NUMBERS,

```
←(ELT numbers 3)
3
←(ELT numbers 6)
ILLEGAL ARG
```

11.2.7 Copying Arrays

You may copy the contents of an array, as opposed to creating a new pointer to it, by using the **COPYARRAY** function. It takes the form

Function:	COPYARRAY
# Arguments:	1
Arguments:	1) an array address, ARRAYPTR
Value:	An array pointer to the new array whose contents are an exact copy of the argument.

COPYARRAY creates a new array of the same size and type as its argument. Its value is a pointer to the new array. It generates an error message ARG NOT ARRAY if its argument is not an array.

Consider the example

```
←(SETQ more-numbers (COPYARRAY numbers))
{ARRAYP}#5,33100
←(ELT more-numbers 4)
4
```

but note that the two arrays are not EQUAL (since EQUAL does not perform these comparisons):

```
←(EQUAL numbers more-numbers)
NIL
```

11.2.8 Comparing Two Arrays

Two arrays may be tested for equality using EQUAL. EQUAL determines equality by determining if they have the same address. However, EQUAL does not descend into the two arrays to determine if the corresponding elements are equal. Thus, two arrays may be *equal* but not have the same address. Let us define a function **EQARRAYP** that determines if two arrays are equal according to the following criteria:

The two arrays have the same length
The two arrays have the same type of elements
The two arrays have the same origin
The two arrays have equal elements in each position

The first three criteria are relatively easy to check and will probably account for most of the work performed by our function.

EQARRAYP takes the following form:

```
Function:       EQARRAYP
# Arguments:    2
Arguments:      1) an array, ARRAY1
                2) an array, ARRAY2
Value:          The address of ARRAY1 if the two arrays
                are equal; otherwise NIL.
```

A Definition for EQARRAYP

We might define EQARRAYP as follows:

```
(DEFINEQ
    (eqarrayp (array1 array2)
        (COND
            ((OR
                (NOT (ARRAYP array1))
                (NOT (ARRAYP array2)))
                (*
                    If one of the arguments is not
                    an array, return NIL.
                )
                NIL)
            ((NEQ
                (ARRAYSIZE array1)
                (ARRAYSIZE array2))
```

```
        (*
             If the size of the arrays is not
             equivalent, return NIL.
        )
        NIL)
   ((NEQ
        (ARRAYTYP array1)
        (ARRAYTYP array2))
        (*
             If the type of the arrays is not
             equivalent, return NIL.
        )
        NIL)
   ((NEQ
        (ARRAYORIG array1)
        (ARRAYORIG array2))
        (*
             If the origin of the arrays is
             not equivalent, return NIL.
        )
        NIL)
   (T
        (PROG NIL
             (FOR I
                  FROM (ARRAYORIG array1)
                  TO (SUB1 (ARRAYSIZE
                  array1))
                  DO
                     (IF
                        (NOT
                           (EQP
                             (ELT array1 I)
                             (ELT array2 I)))
                          THEN (RETURN NIL)))
             (RETURN array1))))
))
```

Note: EQARRAYP is not a standard function in INTERLISP, but one that you can easily define and save.

11.3 HASH ARRAYS

A *hash array* is an array where information is referenced by a *hash item* rather than a strict numeric index. The association between a hash item and the data it

refers to is called a *hash link*. To use a hash array, INTERLISP computes an address, the *hash address*, in an array, called the *hash array*. At that location is stored the *hash value*, a pointer to the actual value of the data. The contents of a cell in the array are the hash item and the hash value, which together form a hash link.

A hash array is used when the potential universe of items to be represented is large but the actual number of items to be stored and retrieved is rather small. Multiple item values may hash to a single cell in the array. It is assumed that the collision between keys is minimal; otherwise, a regular array representation might be more profitably used.

When an item is hashed, the resulting hash address may already contain a hash link. INTERLISP determines if the entry was derived from the item just hashed. If so, the new value replaces the current contents of the cell. Otherwise, a new address is generated. This process repeats until an empty cell is found in which to place the hash link. When a hash array is seven-eighths full, it is either enlarged or an error is generated.

Retrieving an item works in a similar fashion. A hash item is used to compute an address in the hash array. The item is compared against the hash item in the cell. If they match, the hash value is used to retrieve the desired value. Otherwise, a new address is computed and the process repeats until a hash link containing the item is found. If the hashing process generates a cell address whose entry is NIL, then no hash link exists for the item.

INTERLISP provides a system hash array, **SYSHASHARRAY**, for you if you do not wish to create your own. It has an initial size of 512 cells. To use SYSHASHARRAY in the hashing functions, you must specify NIL as the value of the hash array address.

11.3.1 Creating and Testing Hash Arrays

You may create a hash array by executing **HARRAY**. It takes the form

```
Function:       HARRAY
# Arguments:    1
Arguments:      1) the number of cells, N
Value:          A pointer to the hash array.
```

INTERLISP allocates storage for the hash array and returns a pointer to it of the form {HARRAYP}#x,abcde where the lower-case letters represent the storage address of the hash array. For example,

```
←(SETQ a.hash.array (HARRAY 10))          "INTERLISP-D"
{HARRAYP}#7,1030
```

You may test if a pointer refers to a hash array using the function **HARRAYP**. It takes the form

```
Function:       HARRAYP
# Arguments:    1
Arguments:      1) a hash array address, X
Value:          X, if it is a hash array address.
```

HARRAYP returns the pointer if it indeed points to a hash array; otherwise NIL. For example,

```
←(HARRAYP a.hash.array)
{HARRAYP}#7,1030
```

You may determine the number of cells in a hash array using the function **HARRAYSIZE**. It takes the form

```
Function:       HARRAYSIZE
# Arguments:    1
Arguments:      1) a hash array address, X
Value:          The number of cells in the hash array.
```

Consider the following example:

```
←(HARRAYSIZE a.hash.array)
15
```

INTERLISP-D automatically increases the initial size of the hash array by 50% when it is created. Thus, although I created A.HASH.ARRAY of size 10, INTERLISP-D actually assigned it a size of 15. INTERLISP-10/VAX use a function dependent on the size of the hash array to determine the number of extra cells allocated.

Hash arrays are not implemented in INTERLISP/370.

11.3.2 Storing into and Retrieving from a Hash Array

You may put an item into a hash array using **PUTHASH**. It takes the form

```
Function:       PUTHASH
# Arguments:    3
```

```
Arguments:      1) a key, KEY
                2) a value, VALUE
                3) a hash array address, X

Value:          The new value.
```

PUTHASH computes the hash address from KEY as described above. A hash link from KEY to VALUE is created and placed at the hash address. If a hash link already exists at that hash address, it will be overwritten by the new hash link. You may remove a hash link by specifying VALUE to be NIL. Hash values of NIL are not allowed. For example,

```
←(PUTHASH "steve kaisler" "author" a.hash.array)
"author"

←(PUTHASH "pete rose" "baseball player" a.hash.array)
"baseball player"

←(PUTHASH "vanessa williams" "miss america"
a.hash.array)
"miss america"

←(PUTHASH "rick dempsey" "world series mvp"
a.hash.array)
"world series mvp"
```

KEY may be any type of INTERLISP pointer—atoms, strings, array addresses, lists, etc. If an INTERLISP object other than an atom is used as a key, the exact same item must be used to retrieve the hash value. This is required because INTERLISP compares the hash item stored in the cell with the key to determine if that key produced the hash value. The comparison is performed using EQ.

Clearing a Hash Array

You may clear a hash array that has been partially filled by executing the function **CLRHASH**. It takes the form

```
Function:       CLRHASH

# Arguments:    1

Arguments:      1) a hash array address, X

Value:          The hash array address.
```

CLRHASH removes the hash links in all cells of the hash array. It is a good idea to execute CLRHASH after you have created a hash array before storing the first hash link.

Retrieving an Element from a Hash Array

You may retrieve a value from a hash array using the function **GETHASH**. It takes the form

```
Function:       GETHASH

# Arguments:    2

Arguments:      1) a hash key, KEY
                2) a hash array address, X

Value:          The hash value associated with KEY.
```

GETHASH finds the hash link from KEY. It returns the value associated with the hash value in the hash link. If a hash link does not exist, it returns NIL. For example,

```
←(GETHASH "steve kaisler" a.hash.array)
"author"

←(PUTHASH (LIST 'alex) "the great" a.hash.array)
"the great"

←(SETQ new.key (LIST 'alex))
(alex)
```

which is a new list different from the one used in PUTHASH. So,

```
←(GETHASH new.key a.hash.array)
NIL
```

because the value of NEW.KEY is a different list from that used to create the hash link.

Enlarging a Hash Array

When a hash array becomes seven-eighths full, it may overflow (see Section 11.3.5) or it may generate an error when you attempt to put a new value into it. You can catch the error using ERRORSET. If you do not wish to provide an overflow capability, you may expand the size of your hash array by copying it to another hash array of a larger size. **REHASH** hashes all items in one hash array into a new hash array. It takes the form

```
Function:       REHASH

# Arguments:    2

Arguments:      1) an old hash array address, OLDHARRAY
                2) a new hash array address, NEWARRAY

Value:          The new hash array address.
```

Consider the following example:

```
←(SETQ a.new.hash.array (HARRAY 20))
{HARRAYP}#1,2310

←(REHASH a.hash.array a.new.hash.array)
{HARRAYP}#1,2310

←(GETHASH "vanessa williams" a.new.hash.array)
"miss america"
```

11.3.3 Applying a Function to a Hash Array

MAPHASH allows you to apply a function to each hash link in a hash array. It takes the form

```
Function:       MAPHASH

# Arguments:    2

Arguments:      1) a hash array address, X
                2) a mapping function, MAPHASHFN

Value:          The hash array address.
```

MAPHASHFN is a function of two arguments: the hash value and the hash item. For each hash link in X, MAPHASHFN is applied to the hash value and the hash item. For example, to prettyprint the entire contents of a hash array, we might define the function

```
(DEFINEQ
     (pp.hash.array (hasharray)
          (MAPHASH hasharray
                   '(LAMBDA (hashvalue hashitem)
                            (TERPRI)
                            (PRINTDEF hashitem)
                            (TERPRI)
                            (PRINTDEF hashvalue)))
     ))
```

11.3.4 Dumping Hash Arrays

You may dump a hash array using the function **DMPHASH**. This function is primarily intended to be used with the File Package to preserve the definitions of a hash array on a file for subsequent loading. It takes the form

```
Function:       DMPHASH

# Arguments:    1-N
```

```
Arguments:      1) the names of hash arrays
Value:          The loadable expressions.
```

DMPHASH is an NLAMBDA, nospread function that prints on the primary output file a set of loadable forms that may be used to redefine a hash array when the file is subsequently loaded.

If there are no arguments to DMPHASH, it assumes that you want to dump the system hash array, SYSHASHARRAY.

Care must be exercised when reloading a hash array. READ creates new structures for each of the items and values that it reads from the file. Thus, all pointers except atoms and small integers will lose their EQ identities although they will retain their EQUAL identities.

For example, to dump the system hash array, which you may use as your default hash array, you would place the following S-expression in your File Package commands:

```
(E (DMPHASH))
```

Consider the following example:

```
←(SETQ a.hash.array (HARRAY 10))
{HARRAYP}#151466
←(PUTHASH (LIST 'ALEX) "the great" a.hash.array)
"the great"
←(DMPHASH a.hash.array)
(RPAQ a.hash.array (HARRAY 11))
(PUTHASH '(ALEX) "the great" NOBIND)
NIL
```

11.3.5 Overflow Handling

When a hash array is created, it is given a definite size. As entries are made into the hash array, the hashing process becomes less efficient because new keys hash to slots that are already occupied by hash links. When a hash array becomes seven-eighths full (87.5%), INTERLISP considers it to be full. Attempting to add another hash link will cause a hash table overflow.

When a hash table overflow condition occurs as the result of PUTHASH, either

1. An error will be generated, or
2. The hash array will be enlarged to accommodate additional keys.

An error results if the last argument to PUTHASH is merely a hash array created by the user. Note if the hash array argument is NIL, SYSHASHARRAY

is used. It is automatically increased by 50% whenever an overflow condition occurs.

To prevent an error, you must tell INTERLISP how to enlarge the hash array when an overflow condition occurs. The last argument to PUTHASH takes the alternative form

(⟨hasharray⟩ . ⟨expression⟩)

where ⟨expression⟩ has one of the following values:

⟨integer⟩ A positive integer indicates that a new hash array is created whose size is N cells greater than the old table.

⟨floating #⟩ A floating point number indicates that a new hash array is created whose size is equal to the size of the old table multiplied by the number.

⟨function⟩ A function name or LAMBDA expression that is called upon hash table overflow. It takes one argument—the value of the last argument to PUTHASH. If the function returns a number, that number is used to create a new hash array of the given size. Otherwise, a new hash array is created that is 50% larger than the old hash table.

The function may be used to print a warning message, analyze the hash array and delete some values, or monitor the function (via TRACE, for example) that makes entries.

NIL A new hash array is created that is 50% larger than the old hash array.

For example, assume that A.HASH.ARRAY is seven-eighths full. Then, let us execute the following PUTHASH expression:

```
←(PUTHASH "carter" "ex-president" a.hash.array)
HASH TABLE FULL
{HARRAYP}#7,1030
```

However, if we has used the specification for overflow handling:

```
←(PUTHASH "carter" "ex-president" (CONS a.hash.array 5))
"ex-president"
←(HARRAYSIZE a.hash.array)
20
```

where we remember that even though we specified 10 to HARRAY, INTERLISP-D automatically increased that by 50%.

An alternative might be

```
←(PUTHASH "carter" "ex-president" (CONS a.hash.array 2.0))
"ex-president"

←(HARRAYSIZE a.hash.array)
30
```

Finally, let us look at an alternative using a function

```
(PUTHASH "carter"
         "ex-president"
         (CONS a.hash.array
               '(LAMBDA (x)
                        (PRIN2 "Hash Array Overflow")
                        (TERPRI)
                        (PRIN2 "Increase size by 3")
                        (TERPRI)
                        (ITIMES (HARRAYSIZE a.hash.array)
                                3))))

Hash Array Overflow
Increase size by 3
"ex-president"

←(HARRAYSIZE a.hash.array)
45
```

Note that the following construction causes an error:

```
←(PUTHASH "carter" "ex-president" (LIST a.hash.array 2.0))
undefined function
(2)
```

because INTERLISP expects to find the value in the CDR cell as a result of CONSing the two values together.

11.4 A MATRIX PACKAGE

INTERLISP currently supports one-dimensional arrays, e.g., hash arrays. Many applications require two-dimensional arrays, e.g., matrices, for representing data. This section describes a set of functions that you can use to define and manipulate matrices. These functions are not supported in standard INTERLISP.

11.4.1 Defining a Matrix

A *matrix* is a two-dimensional array, e.g., a array of arrays. The number of rows is independent of the number of columns. A special case is the *square matrix* where the number of rows is equal to the number of columns.

MATRIX creates matrices. It takes the form

```
Function:       MATRIX
# Arguments:    4
Arguments:      1) the number of rows, NROWS
                2) The number of columns, NCOLUMNS
                3) A data type, TYPE
                4) An origin specification, ORIGIN
Value:          An {ARRAYP} address representing the
                storage allocation for the matrix.
```

MATRIX creates an array of NROWS elements. Each of these elements has as its value an array of NCOLUMNS elements. The address of the column arrays are stored as the values of the row elements. The address of the row array is returned as the address of the matrix.

A matrix will have an origin of (0,0) or (1,1) depending on the value of ORIGIN. If ORIGIN is NIL, an origin of (1,1) like that of FORTRAN is assumed. TYPE may be any of the legal values acceptable by ARRAY as specified in the INTERLISP-D manual.

Consider the following example:

```
←(SETQ gnp (matrix 5 2))
{ARRAYP}#7,1044

←(for I from 1 to 5
      do (PRIN1 I) (SPACES 6) (PRINT (ELT gnp I)))
1     {ARRAYP}#7,1064
2     {ARRAYP}#1,11310
3     {ARRAYP}#7,1070
4     {ARRAYP}#7,1104
5     {ARRAYP}#7,1110
NIL
```

where each of the elements of GNP is an array as expected.

A Definition for MATRIX

We might define MATRIX as follows:

```
(DEFINEQ
  (matrix (nrows ncolumns type origin)
    (PROG (address index ilimit)
          (*
               Test for invalid values for number of rows
               and columns.
          )
```

```
        (COND
          ((LEQ nrows 0)
                (ERROR "negative or zero rows"))
          ((LEQ ncolumns 0)
                (ERROR "negative or zero columns")))
        (*
             If no origin is specified, assume (1,1).
        )
        (COND
          ((NULL origin)
                (SETQ origin 1)))
        (AND
             (NOT (IEQP origin 0))
             (NOT (IEQP origin 1))
             (ERROR "bad origin specification"))
        (SETQ address
             (ARRAY nrows 'pointer NIL origin))
        (SETQ ilimit
              (COND
                  ((ZEROP origin)
                       (SUB1 nrows))
                  (T nrows)))
        (SETQ index origin)
loop
        (*
             For each element of the matrix row, create
             an array which represents the columns of
             that row.
        )
        (SETA address
              index
              (ARRAY ncolumns type NIL origin))
        (SETQ index (ADD1 index))
        (AND
             (ILEQ index ilimit)
             (GO loop))
        (RETURN address))
))
```

11.4.2 Getting a Matrix Element

To retrieve a matrix element, you need to specify row and column indices. **ELTM** retrieves a matrix element. It takes the form

```
Function:       ELTM
# Arguments:    3
```

```
Arguments:      1) a matrix address, NAME
                2) a row index, ROW
                3) a column index, COLUMN
Value:          The value stored at the row and column
                entry of the matrix.
```

ELTM generates an error if

1. The row index is less than 0 or greater than the number of rows.
2. The column index is less than 0 or greater than the number of columns.
3. NAME is not the address of a matrix.

Consider the following example:

```
←(ELTM GNP 1 1)
300
```

A Definition for ELTM

We might define ELTM as follows:

```
(DEFINEQ
     (eltm (name row column)
          (AND
               (check.matrix name row column)
               (RETURN NIL))
          (ELT (ELT name row) column)
     ))
```

where CHECK.MATRIX is defined by

```
(DEFINEQ
     (check.matrix (name row column)
          (COND
               ((NOT (matrixp name))
                    (ERROR name "not a matrix")))
          (COND
               ((OR
                    (ILESSP row 0)
                    (IGREATERP row (ARRAYSIZE name)))
                    (ERROR "bad row index"))
               ((OR
                    (ILESSP column 0)
                    (IGREATERP column
                              (ARRAYSIZE (ELT name 1))))
```

```
            (ERROR "bad column index"))
        (T name))
))
```

11.4.3 Setting a Matrix Element

To set a matrix element, you need to specify row and column indices. **SETM** sets the corresponding matrix element. It takes the form

```
Function:       SETM
# Arguments:    4
Arguments:      1) a matrix address, NAME
                2) a row index, ROW
                3) a column index, COLUMN
                4) a value, VALUE
Value:          The new value.
```

SETM operates like ELTM except that it replaces the existing element of the matrix with the new value.

Consider the following example:

```
←(SETM GNP 1 1 300)
300
```

A Definition for SETM

We might define it as follows:

```
(DEFINEQ
    (SETM (name row column value)
        (PROG NIL
            (AND
                (check.matrix name row column)
                (RETURN NIL))
            (SETA (ELT name row) column value)
            (RETURN value))
))
```

11.4.4 Basic Matrix Operations

The basic operations that you may perform on two arithmetic matrices are addition, subtraction, and multiplication. In addition, a matrix may be multiplied by a scalar. These operations are subject to certain conditions. Let M1 and M2 be two matrices with dimensions (i1,j1) and (i2,j2) respectively. Then,

1. M1 and M2 may be added or subtracted provided i1 equals j1 and i2 equals j2.
2. M1 and M2 may be mutiplied if i1 equals j2. The result has dimensions of (i1,j2).

Adding or Subtracting Two Matrices

Given two matrices, M1 and M2, with dimensions (i1,j1) and (i2,j2), respectively, MPLUS and MDIFFERENCE will add or subtract individual elements of the two matrices. They take the form

```
Function:       MPLUS
                MDIFFERENCE

# Arguments:    2

Arguments:      1) a matrix, M1
                2) a matrix, M2

Value:          The address of a new matrix whose elements
                are the sum or difference of the elements
                of M1 and M2.
```

We can define MPLUS and MDIFFERENCE as follows:

```
(DEFINEQ
     (mplus (m1 m2)
          (add.or.subtract.matrices m1 m2 T)
     ))

(DEFINEQ
     (mdifference (m1 m2)
          (add.or.subtract.matrices m1 m2 NIL)
     ))
```

The workhorse function that actually performs the operations is **ADD.OR-.SUBTRACT.MATRICES** which is defined as follows:

```
(DEFINEQ
     (add.or.subtract.matrices (m1 m2 flag)
          (PROG (m3 i1 j1 i2 j2 index)
               (AND
                    (is.matrix m1)
                    (is.matrix m2))
               (SETQ i1 (ARRAYSIZE m1))
               (SETQ i2 (ARRAYSIZE m2))
               (SETQ j1 (ARRAYSIZE (ELT m1 1)))
               (SETQ j2 (ARRAYSIZE (ELT m2 1)))
```

```
        (COND
          ((NEQ i1 i2)
                (ERROR "unequal row dimensions"))
          ((NEQ j1 j2)
                (ERROR "unequal column
                dimensions")))
        (COND
          ((NOT
             (MEMBER
                (ARRAYTYPE (ELT m1 1))
                '(FIXP FLOATP ...)))
             (ERROR m1 "not an arithmetic matrix))
          ((NOT
             (MEMBER
                (ARRAYTYPE (ELT m2 1))
                '(FIX FLOATP ...))
             (ERROR m2 "not an arithmetic
             matrix")))
        (SETQ m3
             (MATRIX i1 j1 (ARRAYTYPE (ELT m1 1))
             1))
    rloop
        (SETQ index j1)
    cloop
        (SETM m3
              i1
              index
              (COND
                (flag
                   (PLUS (ELTM m1 i1 index)
                         (ELTM m2 i1 index)))
                (T
                   (DIFFERENCE (ELT m1 i1 index)
                               (ELT m2 i1
                               index)))))
        (SETQ index (SUB1 index))
        (AND
             (GREATERP index 0)
             (GO cloop))
        (SETQ i1 (SUB1 i1))
        (AND
             (GREATERP i1 0)
             (GO rloop))
        (RETURN m3))
))
```

Note that this definition does not consider whether or not both matrices have the same origin. As an exercise, add the code to test for equivalent origins. You will need to use the ARRAYORIGIN function to determine the origin.

Multiplying Two Matrices

Another operation that is often performed on matrices is multiplication. The usual procedure for multiplying one matrix by another is to compute the dot product of the rows of the first matrix with the columns of the second matrix. Using a transposition procedure, we can turn the columns of the second matrix into rows so that they are aligned with the rows of the first matrix. Then, a dot product procedure may be used on corresponding rows. **MATRIX-MULTIPLY** takes the form

```
Function:       MATRIX-MULTIPLY

# Arguments:    2

Arguments:      1) a matrix, MATRIX1
                2) a matrix, MATRIX2

Value:          A matrix that is the result of multiplying
                MATRIX1 by MATRIX2.
```

MATRIX-MULTIPLY assumes the two matrices are of equivalent dimensions. As an exercise, you might want to recode it to handle two matrices of different dimensions. We might define MATRIX-MULTIPLY as follows:

```
(DEFINEQ
    (matrix-multiply (m1 m2)
        (PROG (m3 i1 j1 i2 j2 rindex cindex)
            (*
                RINDEX is the row index.
                CINDEX is the column index.
                M3 is the resulting matrix.
            )
            (AND
                (is.matrix m1)
                (is.matrix m2))
            (SETQ i1 (ARRAYSIZE m1))
            (SETQ i2 (ARRAYSIZE m2))
            (SETQ j1 (ARRAYSIZE (ELT m1 1)))
            (SETQ j2 (ARRAYSIZE (ELT m2 1)))
            (*
                Check the corresponding dimensions of
                the two matrices.
            )
```

```
      (COND
        ((NEQ i1 i2)
              (ERROR "unequal row dimensions"))
        ((NEQ j1 j2)
              (ERROR "unequal column
              dimensions")))
      (*
           Check that the elements of the matrix
           are numeric so that the arithmetic
           operation will not fail.
      )
      (COND
        ((NOT
           (MEMBER
              (ARRAYTYPE (ELT m1 1))
              '(FIXP FLOATP ...)))
           (ERROR m1 "not an arithmetic matrix))
        ((NOT
           (MEMBER
              (ARRAYTYPE (ELT m2 1))
              '(FIX FLOATP ...))
           (ERROR m2 "not an arithmetic
           matrix")))
        (*
           Transpose m2's columns into rows.
        )
        (SETQ m3 (TRANSPOSE m2))
rloop
      (SETQ rindex 1)
cloop
      (SETQ cindex 1)
eloop
      (SETA m3
            rindex
            cindex
            (DOT.PRODUCT (ELT m1 rindex)
                         (ELT m3 cindex)))
      (SETQ cindex (ADD1 cindex))
      (AND
           (NOT (GREATERP cindex j1))
           (GO eloop))
      (SETQ rindex (ADD1 rindex))
      (AND
           (NOT (GREATERP rindex i1))
           (GO cloop))
```

```
                (RETURN m3))
    ))
```

The TRANSPOSE Function

TRANSPOSE merely exchanges rows for columns in the target matrix. We might define TRANSPOSE as follows

```
(DEFINEQ
    (transpose (m1)
        (PROG (mt i1 j1 rindex cindex)
              (SETQ i1 (ARRAYSIZE m1))
              (SETQ j1 (ARRAYSIZE (ELT m1 1)))
              (SETQ mt
                    (MATRIX i1
                            j1
                            (ARRAYTYPE (ELT m1 1)) 1))
              (SETQ cindex 1)
        rloop
              (SETQ rindex 1)
        loop
              (SETM mt
                    cindex
                    rindex
                    (ELTM m1 rindex cindex))
              (SETQ rindex (ADD1 rindex))
              (AND
                  (GREATERP rindex i1)
                  (GO loop))
              (SETQ cindex (ADD1 cindex))
              (AND
                  (GREATERP cindex j1)
                  (RETURN mt))
              (GO rloop))
    ))
```

Note that this function could have been defined more easily using CLISP constructs (see Chapter 23). As an exercise, you may want to recode this function using CLISP.

The DOT.PRODUCT Function

DOT.PRODUCT merely multiplies elements of vectors together and accumulates their sum. We might define DOT.PRODUCT as follows:

```
(DEFINEQ
    (dot.product (v1 v2)
```

```
(PROG (vsize vsum)
      (SETQ vsize (ARRAYSIZE v1))
      (SETQ vsum 0.0)
vloop
      (SETQ vsum
            (PLUS (ELT v1 vsize)
                  (ELT v2 vsize)))
      (SETQ vsize (SUB1 vsize))
      (AND
          (ZEROP vsize)
          (RETURN vsum))
      (GO vloop))
))
```

Note that DOT.PRODUCT works backward through the two vectors. This eliminates the need for an additional temporary variable.

11.5 SORTING USING ARRAYS

We have already seen a sorting function that works on lists of elements (see Section 6.7). A typical operation that is often performed upon arrays is to sort their contents. In conventional programming languages, arrays are more often used than lists in sorting algorithms. Thus, this section presents a few simple sorting algorithms using arrays.

11.5.1 BubbleSort

The simplest sorting method known is called the *bubblesort* method because items "bubble up" from within the set when it is sorted in ascending order. We imagine an array of elements depicted in a vertical column. Elements with lower key values move up the column to the top of the array. Bubblesorting makes repeated passes over the array. At each pass, two adjacent elements are compared. If they are out of order; that is, the lower-valued one follows the higher-valued one, they are exchanged in place. As a result, after each pass the lowest value has been "bubbled up" to its proper position in the array.

BUBBLE.SORT takes the following form:

```
Function:       BUBBLE.SORT
# Arguments:    1
Arguments:      1) an array to be sorted, XARRAY
Value:          A new array whose contents are sorted in
                ascending order.
```

We might define BUBBLE.SORT as follows:

```
(DEFINEQ
    (bubble.sort (xarray)
        (PROG (xsize element index)
            (*
                INDEX keeps track of the current
                element in the array that we are
                inspecting.
                ELEMENT keeps track of the starting
                element on each pass through the
                array.
            )
            (SETQ xsize (ARRAYSIZE xarray))
            (SETQ element 1)
        loop
            (SETQ index element)
        test
            (*
                Compare the elements at INDEX and
                INDEX+1; if they are out of order,
                swap them. Note that INDEX does not
                exceed the limits of the array
                because of the test performed below.
            )
            (COND
                ((GREATERP (ELT xarray index)
                           (ELT xarray (ADD1 index)))
                    (SWAP xarray index)
                    (SETQ element index)))
            (SETQ index (ADD1 index))
            (*
                Determine if we have completed a pass
                through the array.
            )
            (AND
                (LESSP index xsize)
                (GO test))
            (*
                Determine if sorting is completed.
            )
            (COND
                ((EQUAL element (SUB1 xsize))
                    (RETURN xarray))
                (T
```

```
                    (SETQ element (ADD1 element))))
              (GO loop))
    ))
```

SWAP merely exchanges two elements of the array in place given the index of the first element. We might define SWAP as follows:

```
(DEFINEQ
    (swap (xarray index)
        (PROG (temp)
              (SETQ temp (ELT xarray index))
              (SETA xarray
                    index
                    (ELT xarray (ADD1 index)))
              (SETA xarray
                    (ADD1 index)
                    temp))
    ))
```

This definition of BUBBLESORT sorts a single set of numbers. I encourage you to embellish upon the definition in the following ways:

1. Let the elements of the array be lists. Compare entries in the array based on the CAR of the list (i.e., the "key") and exchange them appropriately.
2. Allow BUBBLESORT to accept a function which is the comparison function for the elements of the array. Since INTERLISP-D arrays can be different datatypes, you may specify different kinds of functions to perform the comparison.
3. Attempt to define a recursive form of BUBBLESORT. Trace the operation of this function to determine its behavior.

11.5.2 Selection Sorting

Selection sorting is an equally simple idea. In the Ith pass through the set of elements, we select the element with the highest key and swap it with the (N-I)th element (where N is the size of the set). As a result, after I passes through the set, the last I elements will be those elements with the highest keys sorted in ascending order. **SELECTIONSORT** takes the following form:

```
Function:       SELECTION.SORT
# Arguments:    1
Argument:       1) an array to be sorted, XARRAY
Value:          The array XARRAY sorted in ascending
                sequence.
```

We might define SELECTION.SORT as follows:

```
(DEFINEQ
    (selection.sort (xarray)
        (PROG (xsize maximum)
              (SETQ xsize (ARRAYSIZE xarray))
        loop
              (SETQ maximum 1)
              (PROG (element)
                    (SETQ element 1)
              loop2
                    (COND
                       ((GREATERP (ELT xarray element)
                                  (ELT xarray xsize))
                            (SETQ maximum element)))
                    (SETQ element (ADD1 element))
                    (AND
                       (LESSP element xsize)
                       (GO loop2))
              (SWAP2 xarray maximum xsize)
              (SETQ xsize (SUB1 xsize))
              (AND
                  (EQP xsize 2)
                  (RETURN xarray))
              (GO loop))
    ))
```

We might define SWAP2 as follows:

```
(DEFINEQ
    (swap2 (xarray max.index last.index)
        (PROG (temp)
              (SETQ temp (ELT xarray max.index))
              (SETA xarray
                    max.index
                    (ELT xarray last.index))
              (SETA xarray last.index temp))
    ))
```

Note that SWAP2 takes two arguments for the locations of the elements to be swapped, whereas SWAP merely exchanges adjacent elements.

12

Mapping Functions

INTERLISP provides two mechanisms for iteratively executing a function while varying the values of its arguments. In Chapter 3, we examined the PROG mechanism which allowed us to develop models for the basic iterative control structures: Unconditional DO, DO...WHILE, and DO...UNTIL. However, we had to explicitly set the new values of the arguments to a function before its next invocation. In this chapter, we shall examine *mapping* functions that apply a given function to successive subsets of its first argument. The form of the subset on each successive iteration is determined by the function.

12.1 GENERIC MAPPING

The basic mapping function is **MAP**. It has the following format

```
Function:       MAP
                MAPLIST
# Arguments:    2-3
Arguments:      1) a list, MAPX
                2) a function, MAPFN1, to be applied to
                   successive subsets of MAPX
                3) a function, MAPFN2, for computing the
                   successive subsets of MAPX
Value:          NIL for MAP;
                a list of values for MAPLIST.
```

MAP operates by applying MAPFN1 to the entire list MAPX, and then to (CDR MAPX) repeatedly until MAPX is exhausted. That is, MAP operates on the successive tails of MAPX. If MAPFN2 is non-NIL, INTERLISP computes

(MAPFN2 MAPX) rather than (CDR MAPX) for the successive subsets. MAPFN1 may take several forms:

1. A LAMBDA expression
2. A FUNCTION expression
3. A QUOTE expression whose argument is a function name or has an EXPR property value.

Because MAP returns NIL as its value, it is primarily used for the side effects generated by MAPFN1. MAPFN1 should do something positive such as setting a flag or changing a data structure when called from MAP.

MAPFN1 and MAPFN2 may be any functions that take one argument. They may take more but, of course, these are assigned the value NIL. In general, MAPFN1 and MAPFN2 should check that their argument is a list before operating upon the argument.

A Definition for MAP

We might define MAP as follows:

```
(DEFINEQ
     (map (mapx mapfn1 mapfn2)
          (PROG NIL
          loop
                (COND
                    ((NLISTP mapx)
                         (*
                              MAP applies only to lists.
                         )
                         (RETURN NIL)))
                (APPLY* mapfn1 mapx)
                (SETQ mapx
                      (COND
                         (mapfn2
                              (*
                                   Use the user-supplied
                                   mapping function to
                                   generate next case.
                              )
                              (APPLY* mapfn2 mapx))
                         (T
                              (CDR mapx))))
                (GO loop))
     ))
```

12.1.1 Returning a List of Values

MAPLIST, an alternative form of MAP, executes exactly like MAP except that it returns a list of the results generated by applying MAPFN1 to MAPX.

We might define MAPLIST as follows:

```
(DEFINEQ
    (maplist (mapx mapfn1 mapfn2)
        (PROG (map.list map.expression)
            (SETQ map.list NIL)
        loop
            (COND
                ((NLISTP mapx)
                    (*
                        MAPLIST works only on
                        lists.
                    )
                    (RETURN map.list)))
            (*
                Results are CONSed to the front of
                the resulting list.
            )
            (SETQ map.expression
                (CONS
                        (APPLY* mapfn1 mapx)
                        map.expression))
            (COND
                (map.list
                    (RPLACD (CDR map.expression)
                            (RPLACD
                            map.expression)))
                (T
                    (SETQ map.list map.expression)))
            (SETQ mapx
                (COND
                    (mapfn2
                        (APPLY* mapfn2 mapx))
                    (T
                        (CDR mapx))))
            (GO loop))
    ))
```

Many alternative forms of MAP, designated in many texts as MAPxxx, are provided by INTERLISP. These are described in the following sections. It is useful to inspect the different definitions of the mapping functions to see how

minor changes in functions can produce new capabilities. We have included definitions of the major MAP functions so that you may inspect them and use them as skeletons for developing more powerful mapping functions.

12.1.2 Mapping on Successive Elements

Many times we want to apply a function to successive elements of a list. INTERLISP provides the function **MAPC** to help us accomplish this task. MAPC applies MAPFN1 to (CAR MAPX) on each iteration. Its value is NIL. If MAPFN2 is non-NIL, it is used in place of CDR to compute the new value of MAPX. It takes the form

```
Function:       MAPC
                MAPCAR

# Arguments:    2-3

Arguments:      1) a list, MAPX
                2) a function, MAPFN1, to be applied to
                   succesive elements of MAPX
                3) a function, MAPFN2, for computing
                   successive subsets of MAPX

Value:          NIL for MAPC;
                a list of values for MAPCAR.
```

An alternative form of MAPC is **MAPCAR**. It executes exactly as MAPC does, but returns a list of the values of the successive invocations of MAPFN1 on the elements of MAPX. Let us inspect how these functions work using MAPCAR:

```
←(SETQ presidents '(tyler polk lincoln hayes))
(tyler polk lincoln hayes)

←(MAPC presidents 'PRINT)
tyler
polk
lincoln
hayes
NIL
```

where NIL is returned as the value of MAPC. Alternatively, by invoking MAPCAR, we obtain

```
←(MAPCAR presidents 'PRINT)
tyler
polk
```

```
lincoln
hayes
(tyler polk lincoln hayes)
```

where the final list is the result of invoking MAPCAR because PRINT returns the value of the object printed.

Because MAPCAR returns a list, it consumes storage to create the list.

A Definition for MAPC

We might define MAPC and MAPCAR as follows:

```
(DEFINEQ
     (mapc (mapx mapfn1 mapfn2)
          (PROG NIL
          loop
                (COND
                    ((NLISTP mapx)
                         (*
                              MAPC works only on lists.
                         )
                         (RETURN NIL)))
                (APPLY* mapfn1 (CAR mapx))
                (SETQ mapx
                      (COND
                         (mapfn2
                              (APPLY* mapfn2 mapx))
                         (T
                              (CDR mapx))))
                (GO loop))
     ))
(DEFINEQ
     (mapcar (mapx mapfn1 mapfn2)
          (PROG (map.list map.expression)
                (SETQ map.list NIL)
          loop
                (COND
                          ((NLISTP mapx)
                         (*
                              MAPCAR works only on lists.
                         )
                         (RETURN map.list)))
                (SETQ map.expression
                      (CONS (APPLY* mapfn1
                                    (CAR map.expression))
                            map.expression))
```

```
                (COND
                    (map.list
                        (RPLACD (CDR map.expression)
                                (RPLACD
                                map.expression)))
                    (T
                        (SETQ map.list map.expression)))
                (SETQ mapx
                      (COND
                          (mapfn2
                              (APPLY* mapfn2 mapx))
                          (T
                              (CDR mapx))))
                (GO loop))
    ))
```

12.1.3 Mapping on Successive Elements: MAPCONC

We noted that MAPCAR always returns a new list containing the results of its execution. Sometimes, we want to modify the list that is presented as an argument. To do so, we use **MAPCON** or **MAPCONC** which NCONC the results of applying MAPFN1 to MAPX onto the original list. They take the form

Function:	MAPCON MAPCONC
# Arguments:	2-3
Arguments:	1) a list, MAPX 2) a function, MAPFN1, to be applied to succesive subsets (MAPCON) or CARs (MAPCONC) of MAPX 3) a function, MAPFN2, to be used to compute the succesive subsets of MAPX
Value:	A list of values of the successive applications of MAPFN1 NCONCed together.

MAPCON computes the same values as MAP/MAPLIST but NCONCs the values to form a list which it returns. MAPCONC computes the same values as MAPC/MAPCAR in the same manner. Consider the following examples:

```
←(SETQ X '(Pelee Etna NIL Vesuvius NIL NIL Krakatoa))
(Pelee Etna NIL Vesuvius NIL NIL Krakatoa)

←(MAPCONC X
          '(LAMBDA (x)
```

```
                    (COND
                        ((NULL x) NIL)
                        (T (LIST x)))))
(Pelee Etna Vesuvius Krakatoa)
```

This MAPCONC expression strips null elements from a list and returns a list of the non-null elements.

```
←(SETQ X
       '((Rome New-York) Milan (Bonn Moscow Paris)
       Lisbon))
((Rome New York) Milan (Bonn Moscow Paris) Lisbon)
←(MAPCONC X
          '(LAMBDA (X)
              (COND
                  ((LISTP x) (APPEND y))
                  ((ATOM x) (LIST x))
                  (T NIL))))
(Rome New-York Milan Bonn Moscow Paris Lisbon)
```

This MAPCONC creates a linear list of all elements in lists or atoms which are present in the argument. Since MAPCONC is NCONCing the results together, it will alter the original input list. To prevent this undesirable side effect, the APPEND expression returns a top-level copy of the argument.

A Definition for MAPCONC

MAPCONC may be defined using several elementary functions of INTERLISP. A possible definition for MAPCONC might be

```
(DEFINEQ
    (MAPCONC (mapx mapfn1 mapfn2)
        (PROG (map1 mape mapy)
        loop
            (COND
                ((NLISTP mapx)
                    (*
                        MAPCONC works only lists.
                    )
                    (RETURN map1))
                ((SETQ mapy
                       (apply* mapfn1 (car mapx)))
                  (COND
                    (mape
                        (RPLACD mape mapy))
```

```
                    (T
                        (SETQ map1
                            (SETQ mape mapy)))))
                (PROG NIL
                loop1
                    (COND
                      ((SETQ mapy (CDR mape))
                            (SETQ mape mapy)
                            (GO loop1)))))
            (SETQ mapx
                (COND
                    (mapfn2
                        (APPLY* mapfn2 mapx))
                    (T
                        (CDR mapx))))
            (GO loop))
    ))
```

All of the mapping functions discussed so far work only on lists as their arguments. These functions return NIL if their argument is not a list. You may consider modifying the function definitions to make a list of the argument if it is not a list. To do so, in each function, you should replace the expression

```
(COND
    ((NLISTP mapx)
        (RETURN ...)))
```

by the expression

```
(COND
    ((NLISTP mapx)
        (SETQ mapx (LIST mapx)))
```

which makes the argument a list if it is not already one.

You may want to define a new set of mapping functions which operate in this manner to complement the basic functions provided by INTERLISP.

12.1.4 Mapping over Two Arguments

As we have seen, the mapping functions described above accept only one argument list to be operated upon. Many times you will want to apply a mapping function to two lists on an element-by-element basis. INTERLISP provides two functions **MAP2C** and **MAP2CAR** that allow you to accomplish this task. Their formats are

```
Function:       MAP2C
                MAP2CAR

# Arguments:    3-4

Arguments:      1) MAPX, an argument list
                2) MAPY, an argument list
                3) MAPFN1, the function to be applied
                4) MAPFN2, the function used to compute
                   successive tails of MAPX and MAPY

Value:          NIL if MAP2C;
                a list of results if MAP2CAR.
```

MAP2C and MAP2CAR operate exactly like MAPC and MAPCAR. However, MAPFN1 is applied to successive elements of MAPX and MAPY. That is, MAPFN1 is a function of at least two arguments. MAP2C (respectively MAP2CAR) terminates whenever one of the two lists is exhausted; that is, when the result of MAPFN2 or CDR (the default) is NIL.

```
←(SETQ numbers-1 '(1 2 3 4))
(1 2 3 4)

←(SETQ numbers-2 '(100 200 300 400))
(100 200 300 400)

←(MAP2CAR numbers-1 numbers-2 (FUNCTION IPLUS))
(101 202 303 404)

←(MAP2CAR numbers-1 NIL (FUNCTION IPLUS))
NIL

←(MAP2CAR numbers-2 numbers-2 (FUNCTION ITIMES))
(10000 40000 90000 160000)
```

A Definition for MAP2CAR

We might define MAP2CAR as follows:

```
(DEFINEQ
     (map2car (mapx mapy mapfn1 mapfn2)
          (PROG (map.list map.expression)
                (SETQ map.list NIL)
          loop
                (COND
                    ((OR
                         (NLISTP mapx)
                         (NLISTP mapy))
```

```
                    (*
                        MAP2CAR works only with
                        lists.
                    )
                    (RETURN map.list)))
            (SETQ map.expression
                (CONS
                        (APPLY* mapfn1
                                (CAR mapx)
                                (CAR mapy))
                        map.expression))
            (COND
                (map.list
                    (SETQ mapx
                        (APPLY* mapfn2 mapx))
                    (SETQ mapy
                        (APPLY* mapfn2 mapy)))
                (T
                    (SETQ mapy (CDR mapy))
                    (SETQ mapx (CDR mapx))))
            (GO loop))
))
```

12.1.5 Mapping Across Atoms: MAPATOMS

In many applications you may want to apply a function to some set of atoms that have been created by your program. One way to do this is to build a list of all the atoms as they are created and apply a function to them using one of the mapping functions. If your application is very large, this may consume substantial space to keep track of all the atoms.

MAPATOMS allows you to apply a function to all the literal atoms in the system. However, its major problem is that it uses both atoms defined or created by INTERLISP itself as well as all the atoms that you have created. In order to apply it to a specific subset of atoms, your function, which is passed as an argument, must perform a filtering of all the atoms to select those to be operated upon.

MAPATOMS takes the following format

```
Function:       MAPATOMS
# Arguments:    1
Argument:       1) a function to be applied, FN
Value:          NIL
```

The value of MAPATOMS is NIL so it is the responsibility of the function to generate a result that you may later utilize. If you want to print all of the atoms in the system (of which there are a great many!), you might use the following expression

```
←(MAPATOMS '(LAMBDA (x) (PRINT x)))
f
/SETPROPLIST
FORKBLOCK
DPROG
CHANGEDFNSLST
DUMPSTATCOMS
DECLARE
INTEGERLENGTH
MERGEINSERT
...
```

which are the first few atoms printed from an INTERLISP-10 system.

Note that USERWORDS (see Section 22.7.2) is a list of all the atoms that you have entered via type-in. You may apply MAPATOMS to this list to operate only upon the atoms that you have created. Unlike other LISP dialects, such as FranzLisp or MACLisp, there is no function corresponding to (OBLIST) which returns a list of all atoms created by the user.

You may print every atom with a function definition using the following expression

```
←(MAPATOMS '(LAMBDA (x)
            (COND
                    ((GETD x) (PRINT x)))))
/SETPROPLIST
FORKBLOCK
DECLARE
INTEGERLENGTH
MERGEINSERT
...
```

If you attach properties to each of the atoms that you create in your program describing the usage of the atom, then you may code a MAPATOMS expression to iterate over selected sets of atoms and perform some special operation on them.

12.1.6 A Generic Printing Function

INTERLISP provides **MAPRINT** as a general printing function. Even though it is an output function, we discuss it in this chapter because it operates as a mapping function.

MAPRINT has the following format

```
Function:       MAPRINT

# Arguments:    7

Arguments:      1) an argument list, MAPX
                2) an output file, FILE
                3) a left expression demarcator, LEFT
                4) a right expression demarcator, RIGHT
                5) an expression separator, SEP
                6) a print function, PFN
                7) LISPXPRINTFLG

Value:          NIL
```

MAPRINT applies PFN to successive elements of MAPX. PFN should be a printing function. If it is NIL, PRIN1 is assumed. The results produced by MAPRINT are directed to FILE. If FILE is NIL, then T is assumed (i.e., the terminal although it may also be explicitly specified).

You may use LEFT, RIGHT, and SEP to construct expression forms to suit your application. Before each expression result from applying PFN to an element of MAPX is printed, MAPRINT will print the value of LEFT. Similarly, after the expression is printed, MAPRINT will print the value of RIGHT. Individual expressions are separated by the value of SEP or " " (a string consisting of a single space) if SEP is NIL.

To mimic the application of PRIN1 for lists, you could execute the following function call

```
←(MAPRINT presidents NIL %( %))
(tyler polk lincoln hayes)NIL
```

at the terminal since the file specification is NIL.

```
←(MAPRINT presidents NIL NIL NIL NIL 'PRINT)
tyler
polk
lincoln
hayes
NIL
←(MAPRINT presidents T NIL '%. '%,)
tyler,polk,lincoln,hayes.NIL
```

because it is using PRIN1 to print the output. We can modify this to place spaces between the entries and print NIL on the following line as follows

```
←(MAPRINT presidents
          T
          NIL
          (CONCAT (MKSTRING (CHARACTER (CHARCODE CR)))
                  (MKSTRING (CHARACTER (CHARCODE LF))))
          ", ")
tyler, polk, lincoln, hayes
NIL
```

If the LISPXPRINTFLG is T, then LISPXPRIN1 will be used in place of PRIN1 (see Section 15.1.1).

12.2 APPLYING FUNCTIONS TO SUBSETS

When we apply a mapping function to a list, we often want the result to reflect only the non-NIL values. In the generic mapping functions described in Section 12.1, a NIL will be explicitly reflected in the resulting list of those functions returning a list. **SUBSET**, which returns only the non-NIL values, takes the form

```
Function:       SUBSET

# Arguments:    2-3

Arguments:      1) a list, MAPX
                2) a function, MAPFN1, to be applied to
                   the successive subsets of MAPX
                3) a function, MAPFN2, to be used to
                   compute the successive subsets of MAPX

Value:          A list of the non-NIL values resulting
                from applying MAPFN1 to MAPX.
```

Consider the following example:

```
←(SUBSET '(a 2 b 6 x 32 i v 14) (FUNCTION NUMBERP))
(2 6 32 14)
```

A Definition for SUBSET

We might define SUBSET as follows:

```
(DEFINEQ
   (subset (mapx mapfn1 mapfn2)
      (PROG (map.list map.expression)
            (SETQ map.list NIL)
      loop
```

```
                (COND
                   ((NLISTP mapx)
                        (*
                             SUBSET works only on lists.
                        )
                        (RETURN map.list))
                   ((APPLY* mapfn1 (CAR mapx))
                        (*
                             If the result is non-NIL, then
                             CONS it to the list of results.
                        )
                        (COND
                           ((NULL map.list)
                              (SETQ map.list
                                    (SETQ map.expression
                                       (CONS (CAR mapx)))))
                           (T
                              (SETQ map.expression
                                    (CDR
                                       (RPLACD map.expression
                                          (RPLACD
                                             (CONS (CAR mapx)
                                             mapexp))))))
                           )))
                   (SETQ mapx
                         (COND
                            (mapfn2
                                 (APPLY* mapfn2 mapx))
                            (T   (CDR mapx))))
                   (GO loop))
        ))
```

SUBSET returns a list that contains only the non-NIL values produced by applying a function to successive elements of a list. SUBSET has the same format as MAPCAR. Unlike MAPCAR, however, NIL values produced when MAPFN1 is applied to MAPX are ignored in constructing the resulting list.

12.3 SPECIFYING AN ARGUMENT AS A FUNCTION: FUNCTION

In many applications we want to pass the name of a function as an argument to another function. INTERLISP provides the **FUNCTION** function to assist us. FUNCTION is an NLAMBDA function that does not evaluate its arguments. It has the following format

```
Function:       FUNCTION

# Arguments:    2

Arguments:      1) a function name or definition, FN
                2) an environment specification,
                   ENVIRONMENT

Value:          Either the function name or a FUNARG
                specification.
```

If the environment specification is NIL, then FUNCTION operates exactly like QUOTE by returning the function name or definition. For example,

```
←(MAPCAR presidents (FUNCTION PRINT))
tyler
polk
lincoln
hayes
NIL
```

yields the same results as

```
←(MAPCAR presidents 'PRINT)
tyler
polk
lincoln
hayes
NIL
```

Note: the difference in the two forms is that when the expression is compiled (see Chapter 31), code will be produced for the first form above while in the second it will not. This is particularly important when the argument to FUNCTION is a LAMBDA expression. In this case, the compiler will define and compile an auxiliary function for the LAMBDA expression, whereas if the LAMBDA expression is the argument of QUOTE, it will not. Thus, it is good programming practice to always specify functions passed as arguments using the FUNCTION mechanism to ensure that the program operates properly in both compiled and interpreted modes.

If the environment specification is not NIL, it may be a list of variables that are used freely by the function specified. By free variables, we mean those variables that are globally accessed by the function but which are not specified in the function's parameter list. In this case, FUNCTION returns a FUNARG specification (see Section 12.4) which defines the stack frame where the values of the variables specified in the environment are bound.

If ENVIRONMENT is an atom, it is evaluated and its value is used as a list of free variables.

ENVIRONMENT may also be a stack pointer (see Chapter 30) which points to a stack frame containing bindings for the variables used by by FN.

12.4 THE FUNARG MECHANISM

When FUNCTION is given an environment specification, INTERLISP must know where the values of the variables are to be found before it can execute the function. To do so, it constructs a **FUNARG** expression that has the form

```
(FUNARG function stack-position)
```

where STACK-POSITION is a pointer to a stack frame (see Chapter 30) that contains the bindings of the variables specified in the environment.

FUNARG is not a function, but like LAMBDA and NLAMBDA, it has a special meaning that is recognized by INTERLISP when applying a function to a set of arguments. The virtue of the FUNARG mechanism is that it allows the user to define the environment in which the variables are bound rather than INTERLISP itself (which always uses the most recent stack frame).

The FUNARG mechanism is an advanced feature of INTERLISP that requires considerable caution in its usage. You probably should read and understand Chapter 30 concerning the stack before you attempt to use FUNARGs directly.

12.4.1 Using FUNARGs

Suppose that you have a function that you would like to apply to an argument and also apply to its result. The IRM [irm83] suggests an example (after which the following is modeled)

```
←(DEFINEQ
    (reapply (fn value)
        (APPLY* fn (APPLY* fn value))
    ))
(reapply)
```

REAPPLY applies FN to VALUE, and then applies FN to the result. That is, it computes the composition of a function with itself (in mathematical terms). Consider the example

```
←(REAPPLY (FUNCTION (LAMBDA (X) (ITIMES X X))) 5)
625
```

Let us trace this activiy to see what happens (tracing is defined in Section 20.1):

```
←(TRACE ITIMES)
(ITIMES)

←(REAPPLY (FUNCTION (LAMBDA (X) (ITIMES X X))) 5)
ITIMES:
*ARG1* = 5
*ARG2* = 5
U = 2
ITIMES = 25

ITIMES:
*ARG1* = 25
*ARG2* = 25
U = 2
ITIMES = 625

625
```

which is the answer we expect. However, suppose we execute the following expression

```
←(SETQ value 10)
10

←(REAPPLY (FUNCTION (LAMBDA (X) (ITIMES X VALUE))) 5)
ITIMES:
*ARG1* = 5
*ARG2* = 5
U = 2
ITIMES = 25

ITIMES:
*ARG1 = 25
*ARG2* = 5
U = 2
ITIMES = 125

125
```

which is not what we expect to see! This should multiply the value of the global variable VALUE by the argument to the LAMBDA expression. The problem is that the LAMBDA expression is treated like an EXPR. VALUE is bound within the context of REAPPLY to its second argument. The solution (as noted by the IRM) is to pass the value of VALUE as an environment for the EXPR. This expression would take the form

```
←(REAPPLY
    (FUNCTION (LAMBDA (X) (ITIMES X VALUE))
              (VALUE))
    5)
ITIMES:
*ARG1* = 5
*ARG2* = 10
U = 2
ITIMES = 50

ITIMES:
*ARG1* = 50
*ARG2* = 10
U = 2
ITIMES = 500

500
```

which is the correct answer. Note that VALUE now has the correct binding (i.e., 10) because the second argument to FUNCTION specifies an environment in which its value is accessible.

As the IRM notes [irm83], this example is somewhat contrived to force the problem to be demonstrated—namely, the clashing of variable names. Of course, we could fix the problem by changing the names of the arguments so that they don't conflict. However, in large systems with hundreds of functions, you may not notice problems of this sort until the functions are strongly embedded in the system. It may not be feasible to edit all of the functions to change argument names (even with Masterscope). The method described above provides a fix that allows you to continue working until you are ready to rewrite the affected portions of the system.

12.4.2 Constructing FUNARGs to be Passed as Functions

The IRM notes that you may also construct a FUNARG which may be passed to different functions for application. The FUNARG resides on the stack, and so consumes no permanent storage. Consider the following example (after [irm83]:

```
←(DEFINEQ
    (make.counter (count)
        (FUNCTION
            (LAMBDA NIL
                (PROG1 count
                    (SETQ count (ADD1 count))))
            (count))
    ))
(make.counter)
```

```
←(SETQ counter-1 (MAKE.COUNTER 1))
(FUNARG (LAMBDA NIL (PROG1 COUNT (SETQ COUNT (ADD1
COUNT))))
[STACKP]#152733/FUNARG)
```

So, MAKE.COUNTER returns a FUNARG that increments the counter and returns the previous value of the counter. Each call to MAKE.COUNTER creates a new FUNARG expression with a new, independent environment. Thus, multiple counters may be generated and used throughout a program.

Now, to apply the FUNARG, we can execute the expression

```
←(APPLY counter-1)
1

←REDO                    "this is a Programmer's Assistant
2                         command (see Chapter 25)"
```

Note that the counter's previous value is maintained in the stack frame, so that each successive application of the counter produces the correct value.

Let us create a second counter, which is initialized to count from 100:

```
←(SETQ counter-2 (MAKE.COUNTER 100))
(FUNARG (LAMBDA NIL (PROG1 COUNT (SETQ COUNT (ADD1
COUNT))))
[STACKP]#152732/FUNARG)
```

This is an independent environment (note change in stack address). Now, let us apply our new counter:

```
←(APPLY counter-2)
100

←REDO
101
```

and, for good measure, let us check to see that COUNTER-1 is independent:

```
←(APPLY counter-1)
3
```

Thus, by using a FUNARG expression, you can create a function object (something that acts like a function, looks like a function, but is not a function) which has updatable bindings and which maintains the value of those bindings between calls to the function object. The bindings are only accessible through an instance of that function object (i.e., when you actually execute the function object)

```
←count
UNBOUND ATOM
COUNT
```

You may want to compare and contrast this mechanism with the notion of generators which was introduced in Section 8.11.

12.5 APPLYING A FUNCTION TO ITS ARGUMENTS

When we pass a function name as an argument, it is usually accompanied by a list of variables that it will operate on. The function, however, expects its arguments to be enumerated such that there is one-to-one correspondence between the arguments and the parameters. For example, suppose we have a list of numbers that we want to add together:

```
←(SETQ numbers '(10 23 35 46))
(10 23 35 46)
```

We cannot say (PLUS numbers) because PLUS expects a set of individual numbers to work with, e.g.,

```
←(PLUS 10 23 35 46)
114
```

Thus, (PLUS numbers) would result in an error because the argument is not an atom. PLUS does not know how to decompose a list of numbers into the individual numbers that it will compute with. We, of course, could enumerate the list, e.g., (PLUS (CAR numbers) (CADR numbers) (CADDR numbers) (CADDDR numbers)), but this is both tedious and quickly runs into problems if the list is longer than four items.

INTERLISP provides the function **APPLY** to deal with this problem (so that we do not have to do the enumeration!). Its format is

```
Function:       APPLY
# Arguments:    2
Arguments:      1) a function
                2) a list of arguments
Value:          The value of the function operating upon
                the argument list
```

Thus, we can say

```
←(APPLY 'PLUS numbers)
114
```

or better yet

```
←(APPLY (FUNCTION PLUS) numbers)
114
```

APPLY performs the appropriate interfacing between the function and the list of arguments that we would like it to operate upon.

APPLY does not evaluate the elements of the argument list. However, the arguments given to APPLY are evaluated because it is a LAMBDA function. The function is responsible for evaluating the arguments that appear in the argument list. For example,

```
←(APPLY (FUNCTION SETQ)
        '(FOO (ADD1 3)))
4
```

which sets FOO to 4, but

```
←(APPLY (FUNCTION SET)
        '(FOO (ADD1 3)))
(ADD1 3)
```

will set FOO to (ADD1 3).

An alternative form, **APPLY***, collects an indefinite number of arguments (2-N) into a list so that the function may operate upon them. For example,

```
←(APPLY* (FUNCTION PLUS) 10 23 35 46)
114
```

which is equivalent to

```
(APPLY (FUNCTION PLUS)
       (LIST 10 23 35 46)) → 114
```

APPLY and APPLY* have also been described in more detail in Chapter 8 because they are actually functional forms. Please consult Section 8.9 for additional information.

12.5.1 Determining S-expression Depth

As we have seen, S-expressions may be complex structures. Suppose that we wanted to know the depth of an S-expression, e.g., the level of the deepest sublist of the given S-expression. We might define a function DEPTH that calculates the depth as follows:

```
←(DEFINEQ
    (depth (s-expression)
        (COND
            ((NULL s-expression) 1)
            ((ATOM s-expression) 0)
            (T
               (ADD1
                 (APPLY (FUNCTION MAX)
                        (MAPCAR s-expression
                        'DEPTH)))))
    ))
```

DEPTH returns 1 if the S-expression is NIL because we treat NIL as the empty list. It returns 0 if the S-expression is an atom because it is not a list at all. Otherwise, we apply DEPTH recursively to all of the elements of the S-expression.

13

Arithmetic Functions

As we noted in Chapter 2, there are three different types of numbers in INTERLISP: small and large integers, and floating point numbers. The minimum and maximum values are stored in system variables. For INTERLISP-D, the following ranges are observed

small integers	MIN.SMALLP	-65536
	MAX.SMALLP	65535
large integers	MIN.FIXP	-2147483648
	MAX.FIXP	2147483647
floating point	MIN.FLOAT	-3.402823 E38
	MAX.FLOAT	3.402823 E38

INTERLISP-10 (version dated 26-SEP-83) does not have these system variables defined as of this writing.

INTERLISP/370 maintains the same values for large integers and floating point numbers, but treats small integers as 24-bit quantities. Thus, the range for small integers in INTERLISP/370 is [-2**24,2**24].

A *small integer* is one which satisfies the predicate SMALLP, but it is also an integer which may be stored in the pointer field of a word.

Large integers or floating point numbers are stored as full word quantities. To distinguish these numbers from other INTERLISP pointers, INTERLISP "boxes" the number. That is, when a large integer or floating point number is created either by an arithmetic operation or via READ, it is stored into a full word. A pointer to that word is passed around among functions rather than the actual numeric quantity itself. We say that the number has been *boxed*. When some arithmetic function requires the actual "value," the converse function of *unboxing*, i.e., performing the extra level of addressing, retrieves the number. Boxing consumes two words of storage for each number that is created: one for the number and one for the pointer. Unboxing requires no additional storage.

13.1 INTEGER FUNCTIONS

INTERLISP provides the standard arithmetic functions for computing on integers, both small and large. Integer functions work *only* on integers. If they are given a floating point number, it is first truncated to form the corresponding integer and then used in the computation. If an integer function is given a non-numeric argument, it will display an error message: NON-NUMERIC ARG.

13.1.1 Integer Addition

You can add two or more integers using **IPLUS**. It takes the form

```
Function:       IPLUS
# Arguments:    1-N
Arguments:      1) a number, X1
                2) a number, X2
                3-N) numbers, X3 ... XN
Value:          The integer sum of the numbers Xi.
```

IPLUS is a nospread function, meaning it may take an indefinite number of arguments. Consider the following examples:

```
←(SETQ number-of-oranges 10)
10
←(SETQ number-of-apples 20)
20
```

where the numbers are treated as literal atoms (as described in Chapter 2). We can then add the number of apples and oranges by

```
←(IPLUS number-of-apples number-of-oranges)
30
```

IPLUS is commutative, so we can reverse the order of the arguments and obtain the same answer

```
←(IPLUS number-of-oranges number-of-apples)
30
```

IPLUS treats NIL as zero. Thus, we can use either of the following forms

```
←(IPLUS)
0
```

```
←(IPLUS 5 10 NIL 15 20 NIL)
50
```

where the last example demonstrates that IPLUS may take multiple arguments.

Adding One to a Number
An alternative function, **ADD1**, provides for the very common operation of adding one to a number. It takes the form

```
Function:       ADD1
# Arguments:    1
Argument:       1) a number, X1
Value:          The number plus 1 as an integer.
```

Consider the example

```
←(ADD1 number-of-oranges)
21
```

A Definition for ADD1
We might define ADD1 as follows:

```
(DEFINEQ
     (add1 (x)
          (IPLUS x 1)
     ))
```

13.1.2 Integer Subtraction

Integer subtraction is provided by the **IDIFFERENCE**, which subtracts its second argument from its first argument. It takes the form

```
Function:       IDIFFERENCE
# Arguments:    2
Arguments:      1) a number, X1
                2) a number, X2
Value:          The difference between X1 and X2 as an
                integer.
```

Consider the following examples:

```
←(IDIFFERENCE number-of-oranges number-of-apples)
10
```

```
←(IDIFFERENCE)
"Non-Numeric Arg"
←(IDIFFERENCE number-of-apples number-of-oranges)
-10
```

Note that IDIFFERENCE is not a commutative function; that is, the order of the arguments matters in computing the result.

A Definition for IDIFFERENCE
We might define IDIFFERENCE as follows:

```
(DEFINEQ
     (idifference (x y)
          (IPLUS x (IMINUS y))
     ))
```

Subtracting One from a Number
A special function is provided for the common operation of subtracting one from a number, **SUB1**. It takes the form

```
Function:       SUB1
# Arguments:    1
Argument:       1) a number, X1
Value:          The value of X1 minus 1 as an integer.
```

Consider this example:

```
←(SUB1 number-of-apples)
9
```

A Definition for SUB1
We can define SUB1 in terms of IDIFFERENCE as follows:

```
(DEFINEQ
  (SUB1 (a-number)
     (COND
          ((NUMBERP a-number)
               (IDIFFERENCE a-number 1))
          (T
               (ERROR "Non-numeric Argument")))
  ))
```

or, alternatively,

```
(DEFINEQ
     (sub1 (x)
          (IPLUS x -1)
     ))
```

Negating a Number
Negation of a number is often treated as the subtraction of that number from zero. INTERLISP provides the **IMINUS** function to negate a number. It takes the form

```
Function:       IMINUS
# Arguments:    1
Argument:       1) a number, X1
Value:          The negative value of the number.
```

Consider the example

```
←(IMINUS number-of-oranges)
-20
```

A Definition for IMINUS
We might define IMINUS as follows:

```
(DEFINEQ
     (iminus (x)
          (IDIFFERENCE 0 x)
     ))
```

The foregoing examples demonstrate that an INTERLISP kernel actually needs very few primitive arithmetic functions. From these few primitives, all other arithmetic functions may be constructed. Selection of the few primitives should be based on the efficiencies found in the underlying machine architecture.

13.1.3 Integer Multiplication

Integer multiplication is performed by the **ITIMES**. It takes an indefinite number of arguments just as IPLUS does. It takes the form

```
Function:       ITIMES
# Arguments:    1-N
```

```
Arguments:      1) a number, X1
                2) a number, X2
                3-N) numbers, X3 ... XN

Value:          The cumulative product of the numbers Xi.
```

ITIMES is a nospread function. Consider the following examples:

```
←(ITIMES number-of-apples number-of-oranges)
200

←(ITIMES)
1
```

because ITIMES assumes that NIL is treated as 1.

13.1.4 Integer Division

Integer division is provided through two functions: **IQUOTIENT** and **IREMAINDER**. They take the form

```
Function:       IQUOTIENT
                IREMAINDER

# Arguments:    2

Arguments:      1) a number, X1
                2) a number, X2

Value:          The quotient or remainder of X1 and X2,
                respectively.
```

Given two numbers, IQUOTIENT produces their truncated quotient. For example,

```
←(IQUOTIENT number-of-oranges number-of-apples)
2

←(IQUOTIENT number-of-apples number-of-oranges)
0
```

because IQUOTIENT is not commutative.

```
←(IQUOTIENT)
"Non-Numeric Arg"
NIL
```

Meanwhile, IREMAINDER produces the remainder when the first argument is divided by the second. For example,

```
←(IREMAINDER number-of-oranges number-of-apples)
0
←(IREMAINDER number-of-apples number-of-oranges)
10
←(IREMAINDER)
"Non-Numeric Arg"
NIL
```

Dividing a Number by Two

A frequent arithmetic operation that is performed is to divide a number by two. You might define a function called **IHALF** that returns the integer closest to the quotient of the argument and 2:

```
(DEFINEQ
     (ihalf (x)
          (IQUOTIENT x 2)
     ))
```

13.1.5 Minimum and Maximum

INTERLISP provides functions for finding the minimum and maximum of a sequence of integers. These are the **IMIN** and **IMAX** functions, respectively. They take the form

```
Function:       IMIN
                IMAX

# Arguments:    1-N

Arguments:      1) a number, X1
                2) a number, X2
                3-N) numbers, X3 ... XN

Value:          The minimum or maximum number,
                respectively, of the numbers Xi as an
                integer.
```

IMIN and IMAX are nospread functions. Consider the following examples:

```
←(IMIN 7 32 -1 3 568 91)
-1
←(IMAX 7 32 -1 3 568 91)
568
```

Note that (IMIN) will return the value of the smallest possible integer and (IMAX) will return the value of the largest possible integer.

```
←(IMIN)
-2147483648
←(IMAX)
2147483647
```

13.1.6 Integer Modulus

You may compute the integer modulus using **IMOD**, which takes the form

```
Function:       IMOD
# Arguments:    2
Arguments:      1) a number, X1
                2) a number, X2
Value:          The integer modulus of X1 and X2.
```

The integer modulus differs from the value of IREMAINDER in that it always produces a non-negative integer in the range [0...X2]. For example,

```
←(IMOD 42 12)
6
←(IMOD 128 365)
128
←(IMOD)
"non-numeric arg"
NIL
←(IMOD 786 -34)
4
```

13.1.7 Converting to an Integer

You may convert a number of unspecified type to an integer using **FIX**, which takes the form

```
Function:       FIX
# Arguments:    1
Argument:       1) a number, X
Value:          An integer.
```

FIX returns X if it is an integer. Otherwise, it converts X to an integer by truncating the fractional bits of X. Consider the examples

```
←(FIX 10)
10 ?
```

This example causes problems because FIX is also a command to the Programmer's Assistant (see Chapter 25). To properly use FIX from the top level, you must assign its result to a variable.

```
←(SETQ x (FIX 1053.456))
1053
←(SETQ x (FIX))
"non-numeric arg"
NIL
```

13.2 INTEGER PREDICATES

INTERLISP provides a variety of predicates for comparing integers, testing their values and their characteristics.

13.2.1 Boolean Predicates

INTERLISP provides four Boolean predicates for comparing the values of two integers. They take the form

```
Function:       IGREATERP
                ILESSP
                IGEQ
                ILEQ

# Arguments:    2

Arguments:      1) a number, X1
                2) a number, X2

Value:          T, if X1 ⟨relation⟩ X2; otherwise, NIL.
```

The Boolean predicates are defined for two integers [x] and [y] as follows:

```
Predicate           Definition
---------           ----------

IGREATERP           T if [x] > [y]
                    NIL otherwise

ILESSP              T if [x] < [y]
                    NIL otherwise
```

IGEQ	T if [x] =) [y] NIL otherwise
ILEQ	T if [x] (= [y] NIL otherwise

Consider the following examples:

```
←(IGREATERP 75.0 123.4)
NIL
←(ILEQ 24 56)
T
```

If either or both of X1 and X2 is NIL, these Boolean predicates generate an error message "NON-NUMERIC ARG" and return NIL.

13.2.2 Predicates for Testing Equality

INTERLISP provides several predicates for testing for the equality of two integers, or for testing the equality of an integer to an implied value.

To test if two integers are equal, you should use IEQP. It takes the form

```
Function:       IEQP
# Arguments:    2
Arguments:      1) a number, X1
                2) a number, X2
Value:          T, if X1 is equal to X2; otherwise, NIL.
```

IEQP compares the values of the two integers regardless of whether they are large or small. For examples,

```
←(SETQ number-of-hands 2)
2
←(SETQ number-of-legs 2)
2
←(IEQP number-of-hands number-of-legs)
T
```

Note that NUMBER-OF-HANDS and NUMBER-OF-LEGS are both small integers. It is suggested in the IRM [irm83] that you can use EQ if the two arguments are known to be small integers. That is because some implementations of INTERLISP perform a special allocation of small integers to a unique area of

INTERLISP's address space. For reasons of program portability, I suggest that you use the general form, IEQP, which is guaranteed to operate correctly for all implementations even though it may take slightly longer to execute.

A common arithmetic test that is often performed is to test for the equality of an integer to zero. INTERLISP provides the predicate **ZEROP** to perform this test. It takes the form

```
Function:       ZEROP
# Arguments:    1
Argument:       1) a number, X1
Value:          T, if X1 is zero; otherwise, NIL.
```

A Definition for ZEROP
We might define ZEROP as follows:

```
(DEFINEQ
     (zerop (x)
          (IEQP x 0)
     ))
```

Testing for Equality to One
Some implementations of LISP also provide a predicate for testing the equality of an integer to the number 1. Strangely, INTERLISP does not contain such a function. We can easily define such a predicate as follows:

```
(DEFINEQ
  (ONEP (an-integer)
     (COND
          ((NUMBERP an-integer)
               (COND
                    ((FIXP an-integer)
                         (IEQP an-integer 1)))
          (T
               (ERROR "Non-numeric Argument")))
  ))
```

Testing for the Maximum or Minimum Integer
You may also wish to test whether or not a variable has the value of the maximum or minimum integer. We might define these functions as follows:

```
(DEFINEQ
     (maxp (x)
          (IEQP x MAX.FIXP)
     ))
```

```
(DEFINEQ
     (minp (x)
          (IEQP x (MIN.FIXP)
     ))
```

13.2.3 Predicates for Testing Characteristics

INTERLISP provides a number of predicates for testing the characteristics of integers.

Testing For a Negative or Positive Number

We often want to know whether or not an integer is negative. Using ILESSP, we have to evaluate the integer with respect to zero as the second argument. Because this test is so common, INTERLISP provides us with a "hardwired" predicate, **MINUSP**, which takes the form

```
Function:       MINUSP
                PLUSP
# Arguments:    1
Argument:       1) a number, X
Value:          T, if X is negative (i.e., less than 0).
```

MINUSP returns T if its argument X is less than zero, and NIL otherwise. INTERLISP does not provide a corresponding predicate for testing if the value of an integer is positive. However, we can easily define such a predicate as follows:

```
(DEFINEQ
  (PLUSP (an-integer)
     (COND
          ((NUMBERP an-integer)
               (COND
                    ((FIXP an-integer)
                         (IGREATERP an-integer 0))))
          (T
               (ERROR "Non-numeric Argument")))
  ))
```

Testing for a Small Integer

INTERLISP provides a function for testing whether or not an integer is a *small integer* as described above. This predicate is called **SMALLP**, which takes the form

```
Function:       SMALLP
# Arguments:    1
Arguments:      1) a number, X
Value:          T, if X is a small integer.
```

It returns T if its argument lies within the range of small integers for the particular implementation of INTERLISP; otherwise, it returns NIL. The rationale for using small integers (and their special allocation procedures) derived from early implementations of INTERLISP where storage was often limited. More recent implementations use a virtual storage management procedure which, I believe, obviates the need for even considering whether or not an integer is *small*. Therefore, I suggest that the use of SMALLP be foregone.

Testing for an Integer

INTERLISP provides a predicate for determining whether or not its argument is an integer. This function is called **FIXP** and it takes the form

```
Function:       FIXP
# Arguments:    1
Arguments:      1) a number, X
Value:          X, if X is an integer (large or small).
```

Unlike the other integer predicates, FIXP returns the number X if its argument is an integer, but NIL otherwise. It does not generate an error if its argument is non-numeric.

```
←(FIXP)
NIL
```

A Definition for FIXP

We might define FIXP as follows:

```
(DEFINEQ
     (fixp (number)
           (AND (NUMBERP number)
                (NOT (FLOATP number))
                number)
     ))
```

Testing for Even or Odd Integers

INTERLISP provides functions for testing whether an integer is even or odd. They take the form

```
Function:       EVENP
                ODDP

# Arguments:    2

Arguments:      1) a number, X1
                2) a number, X2

Value:          T, if the result is even or odd,
                respectively.
```

EVENP and ODDP are nospread functions. If X2 is not given, EVENP is equivalent to

```
(ZEROP (IMOD X1 2))
```

otherwise, it is equivalent to

```
(ZEROP (IMOD X1 X2))
```

Consider the following examples:

```
←(EVENP 15378)
T
←(ODDP 187237)
T
←(EVENP)
T
←(ODDP)
NIL
←(EVENP 24 4)
T
←(EVENP 128 6)
NIL
←(EVENP 128 -4)
T
```

Note that NIL is treated the same as 0. These examples, taken from a DEC-System-20, show that the system treats 0 as an even number. Caution should be exercised as the notion of even and odd for zero is machine-dependent.

A Definition for EVENP and ODDP

We might define EVENP as follows:

```
(DEFINEQ
     (evenp (x)
          (ZEROP (IMOD x 2))
     ))
```

and ODDP naturally follows as

```
(DEFINEQ
     (oddp (x)
          (NOT (EVENP x))
     ))
```

Testing for a Power of Two

POWEROFTWOP tests if its argument is a power of two. It takes the form

Function:	POWEROFTWOP
# Arguments:	1
Argument:	1) a number, X
Value:	T, if X is equivalent to two raised to a power.

Consider the following examples:

```
←(POWEROFTWOP 1024)
T
←(POWEROFTWOP 103678)
NIL
←(POWEROFTWOP)
NON-NUMERIC ARG
NIL
←(POWEROFTWOP 0)
NIL
←(POWEROFTWOP 1)
T
```

13.3 MANIPULATING INTEGERS

INTERLISP provides several functions that manipulate the values of integer arguments. These functions appear to correspond to instructions that were prevalent in early computers where "bit twiddling" was a major concern of the programmer.

13.3.1 Logical Manipulations

INTERLISP provides three functions that perform logical functions on integers as bit representations. They take the form

```
Function:       LOGAND
                LOGOR
                LOGXOR

# Arguments:    1-N

Arguments:      1) a number, X1
                2) a number, X2
                3-N) numbers, X3 ... XN

Value:          The bitwise logical operation of the Xi as
                an integer.
```

LOGAND, LOGOR, and LOGXOR are nospread functions. The logical operators are defined by the following table:

Function	*Definition*
LOGAND	[x1] and [x2] and ....
LOGOR	[x1] or [x2] or ....
LOGXOR	[x1] xor [x2] xor ....

The value returned by each of these functions is an integer that represents the application of the function to its arguments. For example, if we define the following integers according to their bit representations

27	11011
6	00110
13	01101
10	01010

we obtain the following results:

```
←(LOGAND 27 10)
10
←(LOGAND 6 13)
4
←(LOGOR 27 13)
31
```

```
←(LOGOR 10 27)
27
←(LOGXOR 27 13)
22
```

It seems to me that considering the values of integers as particular bit patterns defeats the purpose for which INTERLISP was developed. These functions appear to be historical artifacts that remain from the days of limited memory.

However, as with conventional programming languages, an argument can be made for the need to manipulate the bits within a word (see bit manipulations below). The problem is that attempting to port an INTERLISP program to a machine with a different word size can be very difficult. As a matter of philosophy, I suggest that you avoid bit manipulation functions unless absolutely necessary.

Logical Negation

INTERLISP-D implements an additional logical operation, **LOGNOT**, which takes the form

```
Function:       LOGNOT
# Arguments:    1
Argument:       1) a number, X
Value:          The logical negation of X.
```

Consider the following examples:

```
←(LOGNOT 27)
-28
←(LOGNOT 0)
-1
```

A Definition for LOGNOT

We might define LOGNOT as follows:

```
(DEFINEQ
     (lognot (x)
          (LOGXOR x -1)
     ))
```

13.3.2 Integer Shift Functions

INTERLISP provides several functions for shifting, both logically and arithmetically, the values of integers when they are treated as bit streams. The general format is

Function:	LSH RSH LLSH LRSH
# Arguments:	2
Arguments:	1) a number, X 2) the number of bits to shift, N
Value:	An integer resulting from bit shift, either arithmetic or logical.

The arithmetic and logical shift functions are defined as follows

LSH	Arithmetic Left Shift
RSH	Arithmetic Right Shift
LLSH	Logical Left Shift
LRSH	Logical Right Shift

Consider the following examples:

```
←(SETQ a-number 32767)
32767
←(LSH a-number 2)
131068
←(LLSH a-number 2)
131068
←(RSH a-number 8)
127
←(SETQ a-number -127)
-127
←(RSH a-number 2)
-32
←(LRSH a-number 2)
1073741792
```

Note that the last example produces a positive number because a logical right shift introduces zeroes in the high order bits.

The basic difference between the logical and arithmetic shift functions lies in their treatment of the sign bit associated with the integer value. For arithmetic right shifts of negative numbers, the sign bit is propagated to each bit location to ensure that the result is a negative number. For logical right shifts, zeroes are

propagated. Logical and arithmetic left shifts always propagate zeroes from the right.

13.3.3 Integer Conversion

INTERLISP provides a function for creating an integer from a floating point number, **FIX**. FIX converts a floating point number to an integer by truncating any fractional bits and generating a new cell with the proper representation. If the given number is already an integer, INTERLISP merely returns the integer value without consuming any additional storage.

The generic format for this function is

```
Function:       FIX
# Arguments:    1
Argument:       a number
Value:          The integer value of the number.
```

Consider the following example:

```
←(FIX 1.2578)
1.2578 ?
←(SETQ y (FIX 1.2578))
1
```

The reason that INTERLISP complains is that FIX is also a command recognized by the Programmer's Assistant. The Programmer's Assistant intercepts type-in via LISPXREAD (see Section 25.1) so that you cannot type FIX directly to the toplevel of INTERLISP.

13.3.4 The Greatest Common Divisor

INTERLISP provides a function, **GCD**, for calculating the greatest common divisor of two integers. Its generic format is

```
Function:       GCD
# Arguments:    2
Arguments:      1) a number, X
                2) a number, Y
Value:          The greatest common divisor of the two
                numbers.
```

Consider the following example:

```
←(GCD 12468 3256)
4
←(GCD 43.54 12.78)
1
```

Note that GCD accepts any two numbers and attempts to find their greatest common divisor. However, this function makes little sense when applied to floating point numbers.

13.4 FLOATING POINT FUNCTIONS

As with integers, INTERLISP also provides functions for operating upon floating point numbers. These functions, when given an integer, convert the integer to a floating point number internally before operating upon it. As with integers, when these functions are given a non-numeric argument, they respond with an error message "Non-numeric Arg"

13.4.1 Floating Point Addition

You may add two or more floating point numbers using **FPLUS**, which takes the form

```
Function:       FPLUS
# Arguments:    1-N
Arguments:      1) a number, X1
                2) a number, X2
                3-N) numbers, X3 ... Xn
Value:          The cumulative sum of the numbers Xi as a
                floating point number.
```

FPLUS is a nospread function. Consider the following examples:

```
←(FPLUS 34.6 123.5 -43.2)
114.9
←(FPLUS)
0.0
```

Adding One to a Number

There is no corresponding floating point function to ADD1 in INTERLISP, but we can easily define one as follows:

```
(DEFINEQ
    (fadd1 (x)
        (FPLUS x 1.0)
    ))
```

13.4.2 Floating Point Subtraction

You may compute the difference between two floating point numbers using **FDIFFERENCE**, which takes the form

```
Function:       FDIFFERENCE
# Arguments:    2
Arguments:      1) a number, X1
                2) a number, X2
Value:          The difference between X1 and X2 as a
                floating point number.
```

Consider the following examples:

```
←(FDIFFERENCE 123.7 98.4)
24.6
←(FDIFFERENCE)
NON-NUMERIC ARG
NIL
```

Subtracting One from a Number

There is no corresponding floating point function to SUB1 in INTERLISP, but we may easily define one as follows

```
(DEFINEQ
    (fsub1 (x)
        (FDIFFERENCE x 1.0)
    ))
```

Floating Point Negation

You may also compute the negative value of a floating point number (equivalent to subtracting it from zero) using **FMINUS**, which takes the form

```
Function:       FMINUS
# Arguments:    1
Argument:       1) a number, X
Value:          The negative value of X as a floating
                point number.
```

Consider the following examples:

```
←(FMINUS 23.6)
-23.6
```

13.4.3 Floating Point Multiplication

You may multiply two or more floating point numbers together using **FTIMES**, which takes the form

```
Function:       FTIMES
# Arguments:    2-N
Arguments:      1) a number, X1
                2) a number, X2
                3-N) numbers, X3 ... xN
Value:          The cumulative product of the numbers Xi
                as a floating point number.
```

FTIMES is a nospread function. Consider the following examples:

```
←(FTIMES 103.5 12)
1242.0
←(FTIMES)
0.0
←(FTIMES 20.3 12.7 0.034)
8.76554
```

13.4.4 Floating Point Division

INTERLISP provides two functions to support floating point division: **FQUOTIENT** and **FREMAINDER**, which take the form

```
Function:       FQUOTIENT
                FREMAINDER
# Arguments:    2
Arguments:      1) a number, X1
                2) a number, X2
Value:          The quotient or remainder, respectively,
                of X1 and X2 as a floating point number.
```

Consider the following examples:

```
←(FQUOTIENT 196.74 23.54)
8.357689

←(FREMAINDER 196.74 23.54)
8.42

←(FQUOTIENT)
"non-numeric arg"
NIL
```

A Definition for FREMAINDER
We might define FREMAINDER as follows:

```
(DEFINEQ
     (fremainder (x1 x2)
          (FDIFFERENCE x1
                       (FTIMES x2
                               (FIX (FQUOTIENT x1 x2))))
     ))
```

13.4.5 Testing Equality of Floating Point Numbers

You may test whether or not two floating point numbers are equal using **FEQP**, which takes the form

```
Function:       FEQP

# Arguments:    2

Arguments:      1) a number, X1
                2) a number, X2

Value:          T, if X1 equals X2 as a floating point
                number.
```

Consider the following examples:

```
←(FEQP 0.0345 0.034)
NIL

←(FEQP)
"non-numeric arg"
NIL
```

Note: this function is defined in INTERLISP-D only, although it is noted in the IRM [irm83] as being available in all versions of INTERLISP.

13.4.6 Floating Point Boolean Functions

INTERLISP provides the corresponding boolean functions for comparing two floating point numbers: **FGREATERP** and **FLESSP**, which take the form

```
Function:       FGREATERP
                FLESSP

# Arguments:    2

Arguments:      1) a number, X1
                2) a number, X2

Value:          T, if X1 is greater than, respectively
                less than, X2.
```

Consider the following examples:

```
←(FGREATERP 278.7 0.0345)
T

←(FLESSP 1.0004 1.00038)
NIL
```

13.4.7 Floating Point Minimum and Maximum

You may determine the minimum or maximum of a sequence of floating point numbers using **FMIN** or **FMAX**, which take the form

```
Function:       FMIN
                FMAX

# Arguments:    1-N

Arguments:      1) a number, X1
                2) a number, X2
                3-N) numbers, X3 ... XN

Value:          The minimum or maximum, respectively, of
                the numbers Xi as a floating point number.
```

FMIN and FMAX are nospread functions. Consider the following examples:

```
←(FMAX 45.12 0.03 67834.23 -3.1415)
67834.23
```

Note that FMIN and FMAX return the smallest and largest floating point numbers, respectively, if their arguments are NIL.

```
←(FMIN)
-3.402823 E38
←(FMAX)
3.402823 E38
```

13.4.8 Converting a Number to Floating Point Format

You may convert a number to floating point format using **FLOAT**, which takes the form

```
Function:       FLOAT
# Arguments:    1
Argument:       1) a number, X
Value:          X, as a floating point number.
```

FLOAT returns X directly if it is a floating point number. Otherwise, FLOAT converts X to a floating point number, e.g.,

```
←(FLOAT 0)
0.0
←(FLOAT (ITIMES 23 78))
1794.0
←(FLOAT)
"non-numeric arg"
NIL
```

13.5 MIXED ARITHMETIC FUNCTIONS

INTERLISP provides a set of generic functions whose result takes on the characteristics of its arguments. If any argument of these functions is a floating point number, all arguments are coerced to floating point format and the result is a floating point number (unless it is T explicitly). Otherwise, these functions act like integer functions. Rather than give a detailed exposition of these functions, the following table depicts the correspondences between these generic functions and those that we have already discussed.

Generic Function	*Integer*	*Floating Point*
PLUS	IPLUS	FPLUS
DIFFERENCE	IDIFFERENCE	FDIFFERENCE
MINUS	IMINUS	FMINUS

TIMES	ITIMES	FTIMES
QUOTIENT	IQUOTIENT	FQUOTIENT
REMAINDER	IREMAINDER	FREMAINDER
GREATERP	IGREATERP	FGREATERP
LESSP	ILESSP	FLESSP
GEQ	IGEQ	-----
LEQ	ILEQ	-----
MIN	IMIN	FMIN
MAX	IMAX	FMAX

13.5.1 Computing the Absolute Value

You may compute the absolute value of a number using the function **ABS**, which takes the form

```
Function:       ABS
# Arguments:    1
Argument:       1) a number, X
Value:          The absolute value of X.
```

ABS returns X if X is greater than zero; otherwise, it returns -X. Consider the following examples:

```
←(ABS 23.0)
23.0
←(ABS 0.0)
0.0
←(ABS -432.76)
432.76
```

The IRM [irm83] notes that ABS uses GREATERP and MINUS to perform its computations.

A Definition for ABS

We might define ABS as follows:

```
(DEFINEQ
     (abs (x)
```

```
        (COND
            ((GREATERP x 0) x)
            (T
                (MINUS x)))
    ))
```

13.6 SPECIAL ARITHMETIC FUNCTIONS

INTERLISP provides a standard set of functions for performing commonly accepted mathematical functions. These include exponentiation, trigonometric operations, logarithms, and random number generation.

13.6.1 Trigonometric Functions

INTERLISP provides the standard trigonometric functions. Each function takes a number, X, and returns its value as a floating point number. Their formats are

```
Function:       SIN
                COS
                TAN

#Arguments:     2

Arguments:      1) a number, X
                2) a flag, RADIANSFLAG

Value:          A floating point number.
```

If RADIANSFLAG is T then X is in radians; otherwise, it is degrees. Consider the following examples:

```
←(for I from 0.0 to 180.0 by 30.0
      do (PRIN1 I) (SPACES 5) (PRINT (SIN I)))
0.0       0.0
30.0      .4997781
60.0      .8656472
90.0      1.0
120.0     .8656472
150.0     .04997779
180.0     -1.87253 E-7
```

where the values of SIN are small integers.

```
←(for I from 0.0 to 180.0 by 30.0
      do (PRIN1 I) (SPACES 5) (PRINT (COS I)))
0.0       1.0
30.0      .8656472
```

```
60.0        .4997781
90.0        0.0
120.0       -.4997784
150.0       -.865647
180.0       -1.0
```

where the values for COS are small integers.

```
←(for I from 0.0 to 180.0 by 30.0
      do (PRIN1 I)(SPACES 5)(PRINT (TAN I)))
0.0         0.0
30.0        .5773462
60.0        1.732063
90.0        3.402823 E38
120.0       -1.732062
150.0       -.5773461
180.0       1.872535 E-7
```

13.6.2 Inverse Trigonometric Functions

INTERLISP also provides standard inverse trigonometric functions. Each function takes a floating point number within the function's input range. Arguments outside the proper range cause an error to be generated. Their formats are

```
Function:       ARCSIN
                ARCCOS
                ARCTAN

# Arguments:    2

Arguments:      1) a number, X
                2) a flag, RADIANSFLAG

Value:          A floating point number.
```

The ranges for input and output for each of these functions are

Function	*Input*	*Output*
ARCSIN	[-1.0, +1.0]	[-90.0, +90.0]
ARCCOS	[-1.0, +1.0]	[180.0, 0.0]
ARCTAN	[-1.0, +1.0]	[134.9998, 45.0]

RADIANSFLAG determines whether the result is returned in radians (T) or degrees (NIL).

Consider the following examples:

```
←(ARCCOS -1.3)
ARCCOS: arg not in range
-1.3
←(ARCCOS 0.5)
60.00001
←(ARCCOS 0.0)
90.0
←(ARCTAN -0.5)
153.4349
←(ARCTAN -1.0)
135.0
←(ARCTAN 1.0)
44.99999
```

Another View of ARCTAN

An alternative form of ARCTAN, **ARCTAN2**, computes the quotient of two numbers, X and Y. ARCTAN2 has the format

```
Function:       ARCTAN2
# Arguments:    3
Arguments:      1) a number, X
                2) a number, Y
                3) a flag, RADIANSFLAG
Value:          A floating point number
```

ARCTAN2 returns a value in the range −180 to +180 degrees, i.e., the result is in the proper quadrant as determined by the signs of X and Y.

Consider the following examples:

```
←(ARCTAN2 -1.0 0.5)
-63.43495
←(ARCTAN2 0.0 1.0)
-180.0
←(ARCTAN2 1.0 0.0)
90.0
```

13.6.3 Exponentiation

EXPT returns the value of its first argument, X, taken to the Nth power. Its format is

```
Function:       EXPT
# Arguments:    2
Arguments:      1) a base, X
                2) a power, N
Value:          An integer or floating point number.
```

If X and N are integers, EXPT returns an integer result; otherwise, it returns a floating point result.

```
←(EXPT 5 2)
25
←(EXPT -3 5)
-243
←(EXPT -0.7 10)
0.2824752
```

If X is negative (X ⟨ 0) and N is fractional (N ⟨ 1.0), EXPT generates an error:

```
←(EXPT -4 0.5)
LOG OF NON-POSITIVE NUMBER
-4
```

If N is floating and either too large or too small, EXPT returns the largest possible floating point value. For example,

```
←(EXPT 150.0 25.0)
3.402823E38
```

13.6.4 Square Root

SQRT returns the square root of its argument, X, as a floating point number. Its format is

```
Function:       SQRT
# Arguments:    1
Arguments:      1) a number, X
Value:          A floating point number.
```

Consider the following examples:

```
←(SQRT 106754.3)
326.7328
←(SQRT 81)
9.0
```

If X is negative (X ⟨ 0), SQRT generates an error:

```
←(SQRT -100.0)
SQRT OF NEGATIVE NUMBER
-100.0
```

SQRT is faster than (EXPT X .5) as the following timing comparisons demonstrate (see Chapter 29 for a description of TIME):

```
←(TIME (SQRT 100))
0 Conses
.173 seconds
10.0
←(TIME (EXPT 100 .5))
0 Conses
.42 seconds
10.00009
```

13.6.5 Logarithms

INTERLISP provides the natural logarithm function and its inverse. Their formats are

Function:	LOG ANTILOG
# Arguments:	1
Arguments:	1) a number, X
Value:	A floating point number.

Consider the following examples:

```
←(LOG 0.0)
LOG OF NON-POSITIVE NUMBER
0.0
←(LOG 10.0)
2.302583
←(LOG 100000.0)
11.51292
```

If X is negative (X < 0), LOG and ANTILOG generate an error message:

```
←(LOG -100.0)
LOG OF NON-POSITIVE NUMBER
-100.0
←(ANTILOG -2.45)
2.000018
←(ANTILOG 0.0)
1.0
←(ANTILOG -0.7)
.08629356
```

13.6.6 Random Number Generation

INTERLISP provides a psuedo-random number generator, **RAND**, and a function for initializing it, **RANDSET**. The format for RAND is

```
Function:       RAND
# Arguments:    2
Arguments:      1) lower bound, LOWER
                2) upper bound, UPPER
Value:          An integer or floating point number.
```

Consider the following examples:

```
←(RAND 1 10)
6
←(RAND 2 100)
42
←(RAND 1000 1050)
1004
←(RAND -150 -175)
-164
```

RANDSET takes the form

```
Function:       RANDSET
# Arguments:    1
```

```
Argument:       1) an initialization value, X
Value:          An internal state.
```

If X is NIL, RANDSET just returns the internal state. For example,

```
←(RANDSET)
(11996 58248 49267 19460 65470 24561 23941 48766 63093
8937 26494 14191 25348 23161 58999 8788 20454 11637 62780
33784 26916 44822 6238 60538 ...)
```

If X is T, the random number generator is initialized using the current value of the system clocks, and RANDSET returns the new internal state. Otherwise, X is interpreted as a previous internal state, i.e., a value of RANDSET, which is used to reset the random number generator. For example,

```
←(SETQ oldstate (RANDSET))
....
←(RANDSET oldstate)
...
```

where the internal state is represented as a long sequence of very large integers (too numerous to display here).

13.7 MORE ARITHMETIC FUNCTIONS

The two major LISP dialects diverged during their formative years. While INTERLISP emphasized programming productivity tools, MACLISP was developed with numerical computation in mind. This split reflected the emphasis of the two communities' research and development interests. Recently, we have seen a trend toward convergence of the two dialects (and other offshoots as well) in COMMONLISP. Neither dialect provides an extensive library of arithmetic functions beyond the normal ones found in programming languages. It is not hard, as this section demonstrates, to develop a fairly extensive set of arithmetic functions.

In this section, we will describe some useful arithmetic functions. Usually, I will define these functions using arrays and CLISP operators. I leave it as an exercise for you to write the definitions that will work if the data structure to be operated upon is a list of numbers.

Note: These functions are not included in the standard INTERLISP as of this writing.

13.7.1 Statistical Functions

A good set of statistical functions should be in the repertoire of every programmer. This section describes some definitions for the most commonly used statistical functions.

Calculating the Mean

Given an arry of numbers, the *mean* or *average* of that array is defined to be the sum of the elements of the array divided by the length of the array. **MEAN** takes the following format:

```
Function:       MEAN
# Arguments:    1
Argument:       1) an array, X
Value:          The mean of the array X.
```

We might define MEAN as follows:

```
(DEFINEQ
     (mean (x)
          (PROG (xsize xorigin xlimit xsum)
                (IF (ARRAYP x)
                    THEN (RETURN NIL))
                xsize←(ARRAYSIZE x)
                xorigin←(ARRAYORIG x)
                (IF (ZEROP xorigin)
                    THEN xlimit←xsize-1
                    ELSE xlimit←xsize)
                xsum←0
                (FOR I FROM xorigin TO xlimit
                     DO xsum←xsum+(ELT x i))
                (RETURN
                    (QUOTIENT xsum xsize)))
     ))
```

Consider the following example:

```
←(SETQ x (ARRAY 5 1))
{ARRAYP}#1,2464
←(FOR I FROM 1 TO 5
      DO (SETA X I (RAND)))
NIL
```

```
←(MEAN X)
38930
```

where the elements of X are 54985, 50146, 17756, 63115, and 8650.

Calculating the Median

The *median* of an array of numbers is the value for which half the values are less than the median and half are greater. Given an array of numbers, we sort it in ascending order and inspect the midpoint to obtain the value of the median. **MEDIAN** takes the form

```
Function:       MEDIAN
# Arguments:    1
Argument:       1) an array, X
Value:          The median of X.
```

We might define MEDIAN as follows:

```
(DEFINEQ
     (median (x)
          (PROG (xsize y)
                (IF "(ARRAYP x)
                    THEN (RETURN NIL))
                y←(ARRAYSORT x)
                xsize←(ARRAYSIZE x)
                (RETURN
                    (ELT x (IQUOTIENT xsize 2))))
     ))
```

Calculating the Variance

The *variance* of an array is defined as the average of the squares of the deviations from the mean. A *deviation* is merely the difference between an individual element of the array and the mean. **VARIANCE** takes the form

```
Function:       VARIANCE
# Arguments:    1
Argument:       1) an array, X
Value:          The variance of X.
```

We might define VARIANCE as follows:

```
(DEFINEQ
    (variance (x)
        (PROG (xmean xsize xorigin xlimit xmeansq
        xsumsq)
            (IF ~(ARRAYP x)
                THEN (RETURN NIL))
            xmean←(MEAN x)
            xmeansq←(TIMES xmean xmean)
            xsize←(ARRAYSIZE x)
            xorigin←(ARRAYORIG x)
            (IF (ZEROP xorigin)
                THEN xlimit←xsize-1
                ELSE xlimit←xsize)
            (FOR i FROM xorigin to xlimit
                DO
                    xsumsq←(DIFFERENCE
                            (EXPT
                                (FLOAT (ELT X I))
                                2.0))
                        xmeansq))
            (RETURN xsumsq))
    ))
```

Calculating the Standard Deviation

The *standard deviation* of an array is just the square root of its variance. Thus, we can define **STANDARD.DEVIATION** as

```
(DEFINEQ
    (standard.deviation (x)
        (IF ~(ARRAYP x)
            THEN NIL
            ELSE (SQRT (VARIANCE x)))
    ))
```

13.7.2 Complex Arithmetic

INTERLISP does not support complex arithmetic (unlike FORTRAN), but we can simulate this feature using the Record Package (see Chapter 27). A complex number will be represented as a dotted pair, but it will be given its own datatype within INTERLISP.

Defining Complex Numbers

We can declare the definition of a complex number using the following expression

```
←(DATATYPE COMPLEX ((REAL FLOATP) (IMAG FLOATP)))
COMPLEX
```

and we may create a complex number using the following expression

```
←(SETQ C1 (CREATE COMPLEX))
[COMPLEX]#170000
```

Note that a new datatype has been created with the name COMPLEX and has been allocated its own storage pool for creating INTERLISP objects of that type.

Let us create a second complex number, C2.

```
←(SETQ C2 (CREATE COMPLEX))
[COMPLEX]#170002
```

But what we would really like to do is capture the creating function plus the setting of initial values in a single function.

A Definition for Defining Complex Numbers

We might define the function COMPLEX as follows:

```
(DEFINEQ
     (complex (r i)
          (PROG (temp)
               (SETQ temp (create COMPLEX))
               (RECORDACCESS 'REAL temp NIL 'REPLACE r)
               (RECORDACCESS 'IMAG temp NIL 'REPLACE i)
               (RETURN temp))
     ))
←(SETQ c3 (COMPLEX 7.5 9.0))
[COMPLEX]#170770
```

Accessing the Real and Imaginary Parts

We may define two functions, **REAL** and **IMAG**, that access the real and imaginary parts of a complex number as follows:

```
(DEFINEQ
     (real (cx)
          (RECORDACCESS 'REAL cx NIL 'FETCH)
     ))
(DEFINEQ
     (imag (cx)
          (RECORDACCESS 'IMAG cx NIL 'FETCH)
     ))
```

```
←(REAL c3)
7.5
←(IMAG c3)
9.0
```

Complex Addition and Subtraction

Complex addition and subtraction are straightforward. We merely add or subtract the corresponding real and imaginary parts of the two complex numbers to produce a new complex number. We can define **CPLUS** and **CDIFFERENCE** as follows:

```
(DEFINEQ
   (cplus (cx1 cx2)
      (PROG (cx3)
            (SETQ cx3
                  (COMPLEX
                        (PLUS (REAL cx1) (REAL cx2))
                        (PLUS (IMAG cx1) (IMAG cx2))))
            (RETURN cx3))
     ))
(DEFINEQ
   (cdifference (cx1 cx2)
      (PROG (cx3)
            (SETQ cx3
                  (COMPLEX
                      (DIFFERENCE (REAL cx1) (REAL cx2))
                      (DIFFERENCE (IMAG cx1) (IMAG cx2))))
            (RETURN cx3))
     ))
←(SETQ c1 (COMPLEX 2.0 2.0))
[COMPLEX]#170002
←(SETQ c2 (COMPLEX 3.0 3.0))
[COMPLEX]#170000
←(SETQ c4 (CPLUS c1 c2))
[COMPLEX]#170752
←(REAL c4)
5.0
←(IMAG c4)
5.0
←(SETQ c5 (CDIFFERENCE c1 c2))
[COMPLEX]#170750
```

```
←(REAL c5)
-1.0
←(IMAG c5)
-1.0
```

Note that I did not check to see if CX1 and CX2 were actually complex numbers before performing the addition upon them. You can easily add the COND clause which checks this and prints the appropriate error message if they are not complex numbers. The conditional part of the COND clause appears as

```
(TYPE? COMPLEX CX1)
```

which returns T if CX1 is a complex number.

Nor did I check if either of the arguments is NIL. If either (or both) argument is NIL, we would like to assume a complex number equivalent to (0.0, 0.0).

Defining Zero as a Complex Number

You may define a function **CZERO** which creates the complex number having both parts equal to 0.0 as follows:

```
(DEFINEQ
     (czero nil
          (COMPLEX 0.0 0.0)
     ))
```

Then, you may modify both CPLUS and CDIFFERENCE to check if either argument is NIL. If so, you merely set the argument (internal to the function) to (CZERO). Here is an example:

```
(COND
     ((NULL cx1)
          (SETQ cx1 (CZERO))))
```

inside of CPLUS before you perform the addition.

Multiplying Two Complex Numbers

Multiplying complex numbers requires additional arithmetic because both real and imaginary parts must be reflected in the final product. Let CX1 and CX2 be complex numbers which have the representation

```
cx1 = (a + bi)          cx2 = (c + di)
```

Then, the product of CX1 and CX2 is

```
cx3 = (cx1 * cx2)
    = (a+bi) * (c+di)
    = ac + adi + bci + bdi**2
    = (ac-bd) + (ad +bc)i
```

We can define CMULT to multiply two complex numbers as follows

```
(DEFINEQ
   (cmult (cx1 cx2)
      (PROG (cx3)
            (SETQ cx3
                  (COMPLEX
                     (DIFFERENCE
                           (ITIMES (REAL cx1)
                                   (REAL cx2))
                           (ITIMES (IMAG cx1)
                                   (IMAG cx2)))
                     (PLUS
                           (ITIMES (REAL cx1)
                                   (IMAG cx2))
                           (ITIMES (IMAG cx1)
                                   (REAL cx2)))))
            (RETURN cx3))
      ))
←(SETQ cx1 (COMPLEX 5.0 5.0))
[COMPLEX]#460000
←(SETQ cx2 (COMPLEX 3.0 3.0))
[COMPLEX]#460002
←(SETQ cx4 (CMULT cx1 cx2))
[COMPLEX]#460012
←(REAL cx4)
0.0
←(IMAG cx4)
30.0
```

You may define a similar function for performing the division of two complex numbers. I leave that as an exercise for the reader.

Printing Complex Numbers

The standard printing functions, when faced with a datatype that they do not know how to print, will merely print its address representation. For example,

```
←(PRINT cx1)
[COMPLEX]#461000
```

However, this does not inform us of the contents of CX1. We can define a method for printing complex numbers using DEFPRINT (see Chapter 15). The declaration takes the form

```
←(DEFPRINT 'COMPLEX (FUNCTION PRINT.COMPLEX))
NIL

←(DEFINEQ
    (print.complex (cx1)
        (LIST NIL (CONS (REAL cx1) (IMAG cx1)))
    ))
(PRINT.COMPLEX)

←(PRINT cx1)
((5.0 . 5.0))
```

13.7.3 Additional Arithmetic Functions

There are many other arithmetic functions which you may easily define that are not included in the standard INTERLISP loadup. This section describes a few of these functions.

Truncation and Rounding

You may want to truncate a number to its integer portion. One approach is to use FIX to yield the integer part of the number. An alternative method uses IQUOTIENT as shown below. Let us define **TRUNCATE** as follows:

```
←(DEFINEQ
    (truncate (x)
        (IQUOTIENT x 1)
    ))
(TRUNCATE)

←(TRUNCATE 5.4)
5
```

You may also want to round a number to the integer nearest a multiple of that number. We might define the function **ROUNDTO** as follows:

```
←(DEFINEQ
    (roundto (x y)
        (TIMES
```

```
                (ROUNDED (QUOTIENT x y))
                y)
    ))
(ROUNDTO)
```

where **ROUNDED**, which returns an integer nearest the argument, is defined as follows

```
←(DEFINEQ
    (rounded (x)
        (TRUNCATE
            (PLUS x
                (QUOTIENT (SIGN x) 2)))
    ))
(ROUNDED)
```

and **SIGN**, which returns the sign of the argument, is defined as

```
←(DEFINEQ
    (sign (x)
        (COND
            ((GREATERP x 0.0) 1)
            ((LESSP x 0.0) -1)
            (T 1))
    ))
(SIGN)
```

We can test these functions as follows:

```
←(SIGN -5.4)
-1
←(ROUNDED -5.4)
-5
←(ROUNDED 10.23)
10
←(ROUNDTO 10.23 6)
1
```

Computing the Reciprocal

The reciprocal of a number is one divided by that number. Obviously, we must check that we are not dividing by zero, and return an error if it is attempted. The main reason for defining **RECIPROCAL** is that gives a cleaner appearance to your functions than repreatedly writing (QUOTIENT 1.0 x). The name RECIP-

ROCAL is easier to understand than the expression above, although they do the same thing.

We might define RECIPROCAL as follows:

```
←(DEFINEQ
    (reciprocal (x)
        (COND
            ((EQP x 0.0)
                (ERROR "Zero has no reciprocal"))
            (T
                (QUOTIENT 1.0 (FLOAT x))))
    ))
(RECIPROCAL)

←(RECIPROCAL 2)
.5

←(RECIPROCAL 45.346)
.02205266

←(RECIPROCAL 0.0)
Zero has no reciprocal
NIL
```

Obtaining the Floor of a Number

The floor of a number is the nearest integer which is less than the number. We might define **FLOOR** as follows

```
←(DEFINEQ
    (floor (x)
        (PROG (truncation)
            (SETQ truncation (TRUNCATE x))
            (COND
                ((GEQ x 0.0)
                    (RETURN truncation))
                ((EQP x truncation)
                    (RETURN truncation))
                (T
                    (RETURN (SUB1 truncation)))))
    ))
(FLOOR)

←(FLOOR 7.345)
7

←(FLOOR -34.98)
-35
```

```
←(FLOOR -12)
-12
```

Note that the reason for the last test in the COND expression above is demonstrated by the two examples using negative numbers. The floor of −34.98 is not −34, because that number is greater than the original argument. On the other hand, the floor of −12 is −12.

Equivalent arguments may be made for constructing a "ceiling" function which has a similar appearance. I leave this as an exercise for the reader.

This demonstrates that new arithmetic functions may be added to your INTERLISP repertoire quite easily. Ideally, you want to collect the functions in a package, using the File Package commands, so that they may be loaded dynamically as needed.

14

Input Functions

INTERLISP provides a rich variety of input functions for bringing data into a program. Input is mediated by read tables that describe how specific characters are to be interpreted. Most of the input functions take both an optional read table and an optional file.

Input is terminated in an INTERLISP symbolic file by the character sequence ⟨CR⟩⟨LF⟩ (e.g., a carriage-return, line-feed). INTERLISP-10 automatically ignores an ⟨LF⟩ appearing after a ⟨CR⟩ in a file. When you type a carriage return at your terminal, INTERLISP-10 automatically inserts a line feed after the ⟨CR⟩.

Note that the internal representation of "end of line" differs between INTERLISP-10 and INTERLISP-D. INTERLISP-D uses the carriage return character (15Q) while INTERLISP-10 uses the EOL character (37Q). By using (CHARCODE EOL) in your programs to look for end-of-line characters, you can make your programs implementation independent, since the EOL entry in the read tables will be set to the appropriate character code.

14.1 READ: THE GENERAL CASE

The most general case that we encounter in INTERLISP is reading S-expressions into a program. **READ** is the function that performs this operation for us. Its format appears as follows:

```
Function:       READ
# Arguments:    1-3
Arguments:      1) a file name, FILE
                2) an optional read table, RDTBL
                3) an optional flag, FLAG
Value:          An S-expression read from the file.
```

READ returns a value when one of two conditions is encountered:

1. A carriage return is encountered in the file or has been typed by the user at the terminal.
2. A right parenthesis is detected that matches a beginning left parenthesis and completes an S-expression.

READ will properly read atoms, numbers, and strings as S-expressions when they are typed at the terminal without enclosing parentheses. Strings are delimited by double quotes as expected. Numbers are interpreted according to the current radix (normally, base 10). Numbers may be entered in a different radix by adding an identifying suffix to the number.

READ echoes the characters that are typed at the terminal back to the terminal. Thus, when you type a carriage return, INTERLISP echoes both a carriage return and a line feed so that new output will appear on the next physical line. If FLAG is T, then the carriage return normally typed after a matching right parenthesis will be suppressed.

READ inputs an S-expression from the specified file. If the file name is NIL or empty, READ assumes that you meant the terminal and reads from T. Note that INTERLISP does not prompt you to enter any data, but merely waits until you type the proper input to proceed.

```
←(SETQ x (READ))
↑
cursor waits here for you to enter data
```

```
←(SETQ x (READ))
1.234⟨CR⟩
1.234

←(NUMBERP x)
1.234
```

In this case, the atom typed in is interpretable as a number, so INTERLISP creates a number. The type of number depends on the nature of the atom that you type. Integers consist of a sequence of numeric characters without a decimal point. INTERLISP also "understands" scientific notation:

```
←(SETQ x (READ))
1E3⟨cr⟩
1000.0
```

The second number printed in the above cases is the result of evaluating the expression, while the first is just INTERLISP echoing what you typed in.

You may specify the radix of the number that you are typing in by appending a radix identifier. INTERLISP assumes all numbers are entered in decimal. Appending a Q to the number indicates an octal number.

```
←(SETQ x (READ))
21Q
17                                  Note: radix change!
```

Because INTERLISP/370 maintains numbers in an internal hexadecimal format (due to the System/370 architecture), prefixing an @ to a number causes it to be interpreted as a hexadecimal number:

```
←(SETQ x (READ))
@11
16
```

Strings are identified to READ by delimiting them by double quotes:

```
←(SETQ x (READ))
"An ill wind blows no good"
"An ill wind blows no good"
```

Input from the terminal is line-buffered in order to permit control characters (principally, backspacing and word delete) to be effective. No characters are passed from the input routines to the program until a ⟨CR⟩ has been typed. But, merely typing a ⟨CR⟩ indicates no input, and INTERLISP waits for further input.

```
←(SETQ x (READ NIL NIL T))
↑
```

typing ⟨CR⟩ does nothing; you must enter) or] to complete an S-expression, but

```
←(SETQ x (READ NIL NIL T))
HELP⟨CR⟩
HELP
←(SETQ x (READ NIL NIL T))
)
NIL
```

because) completes an S-expression, which is in fact the empty list represented by NIL.

14.1.1 Effect of Control Characters

INTERLISP recognizes the effect of several characters (that is, pressing the CTRL key and a letter key simultaneously) when accepting input from the terminal. These *control characters* affect most input functions, although we discuss

them after READ. Note that these control characters are often implementation-dependent. The following control characters are handled by INTERLISP-D and INTERLISP-10:

CTRL-A
Erases the last character typed in by you. It echoes \ and the erased character.

```
←(SETQ x (REED↑A↑AAD))
```

is entered as (SETQ x (READ)) because ED was erased by the pair of CTRL-As that were typed. Note that what you really see on your terminal is

```
←(SETQ x (REED\D\EAD))
```

CTRL-B
Forces a break in the current computation, whence the stack is backed up to the last function call and the BREAK package (see Chapter 20) is entered.

```
←(SETQ x (READ))
↑B
interrupted below \TTYBACKGROUND
(\TTYBACKGROUND broken)
:
```

While in INTERLISP-10, you would see

```
←(SETQ x (READ))
↑B

BREAK
NIL

(READ broken)
:
```

There are significant differences in the way the various implementations handle this interrupt. You should consult the IRM for the current description.

CTRL-C
Forces termination of the current computation and return of control to the operating system.

```
←(SETQ x (READ))
↑C
@                (which is the TENEX prompt)
```

CTRL-D
Forces a termination of the current computation, followed by a reset, and rewinds the stack to the top level. Internally, it forces a call to RESET (see Chapter 18).

CTRL-E
Forces an error in the current computation which is backed up to the last errorset; NIL is returned as its value. Internally, it forces a call to ERROR! (see Chapter 18).

CTRL-H
For INTERLISP-D, a menu of running processes is displayed from which you may select one to be interrupted. This will be discussed in more detail in Volume 2.

For INTERLISP-10, at the next point that a function will be entered, INTERRUPT will be invoked instead. It prints the message "INTERRUPTED BEFORE ⟨fn⟩", constructs an appropriate break expression, and calls BREAK1 (see Section 20.3.3). Break handling is described in Chapter 20. Note that CTRL-H breaks only when a function is to be entered, as opposed to CTRL-B which breaks immediately. It is "safer" to use CTRL-H since the environment is in a relatively clean state from which the computation may be resumed.

CNTRL-O
Clears the current contents of the terminal output buffer.

```
←(SETQ x "Now is the time for all good men to come to
the aid of their country")

←(PRINT x)
"Now is the time for all go↑Ood men to come to the aid of
their country"
"Now is the time for all go
```

The string is placed in the output buffer twice: once, as a result of the PRINT function, and once as a result of the function evaluation. When I interrupted it with a CTRL-O, the first instance had already been passed to PRINT, but the second remained in the output buffer. It is this second instance that was abruptly terminated because the output buffer was cleared.

CTRL-O is not enabled in INTERLISP-D.

CTRL-P
Change the printlevel for the display of S-expressions by clearing and saving the input buffer, clearing the output buffer, and waiting for the user to enter a number indicating the new printlevel. The input buffer is restored and the program continues. The printlevel is described in Section 15.3.

CTRL-Q
Erases the input line buffer contents. It echoes # # and resets the buffer to the last carriage return. Note that you may type ahead several lines but only the last physical record is erased:

```
←(SETQ x (READ NIL↑Q##
```

whence you may start retyping the line. You may not use this character repeatedly.

CTRL-R
Causes INTERLISP to retype the contents of the input buffer. It is used after several editing control characters to see where you are.

```
←(SEL↑ATQ x 9↑(REE↑AD NIL↑R
(SETQ x (READ NIL
                  ↑
                  cursor waits here for you to resume
typing of characters to complete the expression.
```

CTRL-S
Allows you to change the minimum amount of storage to be maintained by the garbage collector. When CTRL-S is typed, the input buffer is cleared and saved, and INTERLISP waits for you to enter a number which is the amount of storage to be maintained. It must be terminated by a period; otherwise, it will be ignored. After the number is entered, the input buffer is restored and computation resumes. Storage management is discussed in Chapter 29.

CTRL-S is not enabled in INTERLISP-D.

CTRL-T
Prints the total execution time for the program as well as its status.

```
←↑T
running in \TTYBACKGROUND in TTWAITFORINPUT in TTBIN
90% UTIL, 2% SWAP, 6% GC
```

And, in INTERLISP-10

```
←↑T
IO WAIT IN READ IN LISPXREAD IN EVALQT, LOAD 1.5
```

CTRL-U
Forces the editor to be invoked for the current expression after it is read (INTERLISP-D).

```
←(SETQ x (READ NIL↑U NIL T))
(edit)
*
```

In INTERLISP-10, this control character erases the input buffer under TOPS-20.

CTRL-V
Allows you to enter an atom from the terminal which has a control character embedded within it. To do so, you type CNTRL-V followed by the character that you wish interpreted as a control character. Typing CNTRL-V C transmits the code equivalent to a CNTRL-C.

CTRL-X
Aborts printout, moves to previous expression and prints it; used in the editor (see Chapter 19).

CNTRL-Y
Returns result of evaluating next expression read as though it had been typed; a READ macro.

CNTRL-Z
Aborts printout, moves to last expression of current expression and prints it; used in editor (see Chapter 19).

14.2 READING: SPECIAL CASES

INTERLISP provides several functions for performing specific input functions. I refer to them as special cases only because they process information in different formats. They should not be construed as being unusual reading mechanisms.

14.2.1 RATOM: Reading an Atom

RATOM allows you to read a single atom from the specified file. Its format is

```
Function:       RATOM
# Arguments:    2
Arguments:      1) a file name, FILE
                2) a read table, RDTBL
Value:          The next atom encountered in the input
                stream.
```

If FILE is NIL, RATOM attempts to read from the current input file. It will return an atom, determined by its PRIN1-PNAME representation, or a number

if one is read. However, RATOM performs no special processing for strings, so if " is encountered in the input stream, it will be returned as the next atom. Consider these examples:

```
←(SETQ x (RATOM))
Come to the high country!
Come

←No do, collect, or join in
(to (the high country))
u.d.f
to
```

because DWIM is enabled (see Chapter 22). Let us disable it and see what happens:

```
←(DWIM NIL)
NIL

←(SETQ x (RATOM))
Come to the high country!
COME

UNDEFINED CAR OF FORM
TO
```

because it returns to the READ-EVAL-PRINT loop. Thus, RATOM should be used primarily within a program where you have control over the input stream.

Note that care must be exercised when reading using RATOM.

```
←(SETQ x (RATOM))
"Baltimore is best"
%"
```

because " is a separator character.

Note that a break or separator character that terminates input to RATOM is not read, but remains in the buffer to become the first character seen by the next input function that is called.

14.2.2 Reading up to an Atom

A variant of RATOM is **RATOMS** which reads atoms from the input file until a specific atom, the first argument, is read. This function allows you to scan the input stream for specific phrases that are prefaced by demarcating atoms. Its format is

```
Function:       RATOMS

# Arguments:    3

Arguments:      1) an atom, ATM
                2) a file name, FILE
                3) A read table, RDTBL

Value:          A list of atoms from the current file up
                to, but not including, the atom ATM.
```

Consider the following example:

```
←(SETQ x (RATOMS '%]))
Washington is a capital city]
(Washington is a capital city)
```

A Definition for RATOMS

We might define RATOMS as follows:

```
(DEFINEQ
     (ratoms (atm file rdtbl)
          (PROG (lst char)
          loop
                (SETQ char (RATOM file rdtbl))
                (COND
                    ((EQ char atm)
                         (RETURN (CAR lst)))
                    ((SETQ lst (TCONC lst char))
                         (GO loop))))
     ))
```

Generally, you may use RATOMS to read text files up to, but not including, punctuation marks. Thus, you can read in phrases and sentences without reading the punctuation marks.

14.2.3 Testing Atom Demarcators

The action of RATOM is terminated by a break or separator character. Sometimes you may want to know what type of character forced the termination of RATOM. **RATEST** allows us to determine the nature of this character. RATEST takes the form

```
Function:       RATEST

# Arguments:    1
```

```
Arguments:      1) a flag, FLAG
Value:          T, if a separator was encountered
                immediately prior to the last atom read.
```

RATEST takes one argument, FLAG, which determines the type of test to be performed. Its values are

Value of FLAG	Type of Test
T	Separator character
NIL	Break character
1	Escape character

RATEST returns T if the character preceding the last atom read was a separator or break character, NIL otherwise. For FLAG=1, RATEST returns T if the last atom contained an escape character. Consider the following examples:

```
←(SETQ FLAG T)
T
←(PROG (TEMP)
       (SETQ TEMP (RATOM))
       (SETQ TEMP (RATOM))
       (RETURN (RATEST FLAG)))
COME TO
T
```

because the space between COME and TO is a separator character.

```
←(SETQ FLAG NIL)
NIL
←(PROG (TEMP)
       (SETQ TEMP (RATOM))
       (SETQ TEMP (RATOM))
       (RETURN (RATEST FLAG)))
COME (TO
T
UNBOUND ATOM
TO
```

because the (is a break character.

See Section 14.4 for a discussion of the read tables and the effects of these different character types.

14.2.4 RSTRING: Reading a String

RATOM treats " as a separator character that terminates the act of reading. Thus, RATOM cannot accept a string as input and return it as a string. If your program expects to receive a string, you should use the function RSTRING which makes a string of the next atom read. **RSTRING** takes the following form:

```
Function:       RSTRING
# Arguments:    2
Arguments:      1) a file name, FILE
                2) a read table, RDTBL
Value:          A string from FILE which is terminated by
                the next break or separator character.
```

The action of RSTRING is terminated by the next break or separator character that is encountered in the input stream. If the next input character happens to be a break or separator character, RSTRING will return " " which is the null string. Your program should explicitly test for the presence of the null string (otherwise program errors may result).

```
←(SETQ x (RSTRING))
"Atlanta"
"Atlanta"

←(SETQ x (RSTRING))
⟨CR⟩
" "

←(SETQ x (RSTRING))
HELP⟨CR⟩
"HELP"
```

The break or separator character terminating a string is not read, but remains in the buffer to be read by the next input function. CTRL-A, CTRL-Q, and CTRL-V all have the same effect as they do on READ.

14.2.5 READC: Reading a Character

Both RATOM and RSTRING return a complete syntactic unit as their value. These syntactic units are terminated by break or separator characters, and their effect is modified by the presence of escape characters (as defined by the read tables). Sometimes you may want to process the input on a character by character basis, perhaps taking special action as individual characters are recognized. One special action might be to read enough characters to recognize a unique command and then complete the rest of the command for the user.

READC reads a single character from a file. It takes the form

```
Function:       READC
# Arguments:    2
Arguments:      1) a file name, FILE
                2) a read table, RDTBL
Value:          The next character in the input stream.
```

READC is unaffected by read tables. That is, it always returns the next character that it encounters in the input stream. READC is subject to line-buffering (see Section 14.6).

```
←(SETQ x (READC))
A
A

←(SETQ x (READC))
(A)
%(
```

14.2.6 Reading the Last Character

Some algorithms, particularly those that process textual material, need to "back up" to see the last character that was read. When READC is executed, the character read is removed from the input buffer. However, INTERLISP allows us to retrieve its value via **LASTC**, which takes the form

```
Function:       LASTC
# Arguments:    1
Arguments:      1) a file name, FILE
Value:          The value of the last character read from
                FILE.
```

LASTC returns the last character read from the file which is its argument. Like READC, LASTC is unaffected by break, separator, or escape characters. Consider the following example:

```
←(PROG (X)
       (FOR I FROM 1 to 5 (SETQ x (READC)) (PRINT x))
       (RETURN (LASTC)))
```

```
WATER⟨CR⟩
W
A
T
E
R
R
```

where the last R is returned as the result of LASTC.

14.2.7 Looking Ahead in the Input Stream

In a similar fashion, you may need to look ahead in the input stream to see what character occurs next. Command or text processors based on LR(k) algorithms often need to "peek" at the next character to determine what action to take with the current character. INTERLISP provides **PEEKC** to look ahead in the input stream. PEEKC takes the following form

```
Function:       PEEKC

# Arguments:    2

Arguments:      1) a file name, FILE
                2) a read table, RDTBL

Value:          The next character in the file.
```

PEEKC returns the next character in the line buffer, but does not actually remove it.

When PEEKC is used with terminal input, it is subject to line buffering. If the value of RDTBL is NIL, PEEKC returns the character as soon as it is typed. Otherwise, it waits until the line has been terminated (see Section 14.6). Consider the following example:

```
←(PROG NIL
       (FOR I FROM 1 TO 5
            (PRIN1 (READC))
            (SPACES 3)
            (PRINT (PEEKC))))
WATER
W   A
A   T
T   E
E   R
R   %
NIL
```

where the % above indicates a null character as the next character in the input buffer.

14.2.8 READLINE: Reading a Terminal Line

Many times the input to a program will be a sequence of atoms that are to be treated as an entire unit. To read this with RATOM or READ would require an iterative loop encoded in your program. INTERLISP provides us with a convenient function, **READLINE**, that gathers a sequence of atoms up to a line terminator and returns it as a list. READLINE takes the following form:

```
Function:       READLINE
# Arguments:    3
Arguments:      1) a read table, RDTBL
                2) a line specifier, LINE
                3) a LISPX flag, LISPXFLAG
Value:          The next line from the terminal, returning
                it as a list.
```

READLINE terminates when any of the following conditions are met:

1. A carriage return (typed by the user) and not preceded by any spaces is encountered.

   If the terminal screen appears as follows:

   ```
   Enter Command: show all paths from main⟨CR⟩
   ```

   READLINE returns the list

   ```
   (SHOW ALL PATHS FROM MAIN)
   ```

   where "Enter command:" was printed by the subsystem as a prompt.

2. A list terminated by], in which case the list is included in the result.

   If the terminal screen appears as follows:

   ```
   Enter Rule: (Implies murder (or death homicide]⟨CR⟩
   ```

   READLINE returns the list

   ```
   (IMPLIES MURDER (OR DEATH HOMICIDE))
   ```

3. An unmatched) or], neither of which is included in the resulting list.
   If the terminal screen appears as follows:

   ```
   ← DF ADD.TO.DOCUMENT.LIST]
   ```

   READLINE returns the list

   ```
   (DF ADD.TO.DOCUMENT.LIST)
   ```

The action of READLINE in case 1 is modified if one or more spaces precedes the carriage return or a list is terminated by a). These spaces indicate that the input stream is to be continued on succeeding lines. READLINE prints "..." and continues reading on the next line.

READLINE is a general reading function that is used by LISPXREAD (see Section 25.5.1). It may be defined as

```
(DEFINEQ
     (readline (rdtbl line lispxflag)
          (APPLY* LISPXREADFN T)
     ))
```

where LISPXREADFN is initially set to READ, but may be redefined by your program.

Consider the following examples:

```
←(SETQ x (READLINE))
]
(NIL)
```

This allows you to type a function name followed by a) or] to LISPX (see Section 25.2). The Programmers Assistant distinguishes between ⟨name⟩] as a function invocation and ⟨name⟩⟨CR⟩ as the evaluation of a variable.

```
←EDITF REAL]
edit
*
```

which allows us to edit the function REAL, because READLINE is used at the top-level READ-EVAL-PRINT loop.

14.2.9 Reading from a File

You may want to read all of the S-expressions in a file without evaluating them. **READFILE** takes the form

```
Function:       READFILE
# Arguments:    1
Argument:       1) a file name, FILE
Value:          A list of all the S-expressions in a file.
```

READFILE reads successive S-expressions from a file until it encounters an end-of-file condition, which causes an error, or it reads the single atom STOP. Let us assume that we have a file named COMPLEX that contains some of the definitions for complex arithmetic that we described in the last chapter. We might read that file as follows:

```
←(READFILE 'COMPLEX)
((FILECREATED "21-JUL-84 10:52:01" ⟨KAISLER⟩COMPLEX..8
3497 changes to: (VARS COMPLEXFNS COMPLEXCOMS) (FNS FLOOR
ROUNDTO SIGN ROUNDEDTO RECIPROCAL TRUNCATE PRINT.COMPLEX
CMULT) (RECORDS COMPLEX) previous date: "17-JUL-84
21:52:16" ...
```

READFILE reads the contents of the file as a single large S-expression. In this format, the S-expression is not very readable (by humans), so we usually pass the result of READFILE to PRINTDEF to prettyprint (see Section 15.7) the output.

```
←(PRINTDEF (READFILE 'COMPLEX))
((FILECREATED "21-JUL-84 10:52:01" ⟨KAISLER⟩COMPLEX..8
3497 changes to:
     (VARS COMPLEXFNS COMPLEXCOMS)
     (FNS FLOOR ROUNDTO SIGN ROUNDEDTO RECIPROCAL
     TRUNCATE PRINT.COMPLEX CMULT)
     (RECORDS COMPLEX)
     previous date: "17-JUL-84 21:52:16" ...
```

where the ... indicates that there is more in the file that we have not shown here.

14.2.10 Skipping S-expressions in a File

You may process input from a file using a combination of several input functions. In some cases, after reading portions of the next S-expression from a file, you may decide not to read the rest of the expression. This often happens when you are parsing input expressions and detect an error early in the expression. Usually, the nature of the error will preclude successful processing of the remainder of the expression.

SKREAD, which allows you to skip expressions within a file, takes the following form:

```
Function:       SKREAD

# Arguments:    2

Arguments:      1) a file name, FILE
                2) an input string, REREADSTRING

Value:          A separator character or NIL.
```

SKREAD has the effect of moving the file pointer ahead in the file to the end of the current S-expression (see Section 16.8). None of the S-expression which is skipped is transferred to memory.

If you have already read some of the expression using READCs or RATOMs, this input should be assigned as the value of REREADSTRING. SKREAD assumes that it has already seen this input and traverses the rest of the expression. REREADSTRING allows you to keep the parenthesis and double-quote counts correct.

SKREAD returns one of three values:

1. %) if the first thing it encounters is a closing parenthesis.
2. %] if the read terminates on an unbalanced], e.g., a] that closes any number of open left parentheses.
3. NIL, otherwise.

SKREAD always uses FILERDTBL, the File Package read table, as its read table.

Let us use the file COMPLEX to demonstrate the application of SKREAD. The first few expressions appear as displayed in Section 14.2.9 above. Thus, by executing the following sequence, we can see how SKREAD works:

```
←(OPENFILE 'COMPLEX 'INPUT 'OLD)
⟨KAISLER⟩COMPLEX..8
```

because we must open the file before we can read from it.

```
←(RATOM 'COMPLEX)
%(

←(RATOM 'COMPLEX)
FILECREATED

←(SETQ REREADSTRING (RATOM 'COMPLEX))
&"

←(SKREAD 'COMPLEX REREADSTRING)
NIL

←(RATOM 'COMPLEX)
⟨KAISLER⟩COMPLEX..8
```

where we have skipped the remaining characters of the date-time group in the file header.

```
←(READ 'COMPLEX)
3497

←(READ 'COMPLEX)
changes

←(SKREAD 'COMPLEX)
NIL

←(READ 'COMPLEX)
(VARS COMPLEXFNS COMPLEXCOMS)
```

and so on. As you can see, SKREAD allows you to selectively read the expressions within a file.

14.3 INPUT PREDICATES

Many programs accept input from different sources such as several files, which may be written during the program's execution, the terminal, or the Ethernet interface. The input functions discussed in the previous two sections all wait for input if none is available when the reading function is executed. If line buffering is activated for the terminal, a program executing a "read" against the terminal must wait until you terminate the line with a carriage return. If you must think about your answer, the program cannot proceed to do anything else. We would like to be able to test the input file to determine if input is available. If none is available, the program can do other useful work. A similar situation applies to reading from files that are dynamically created during program execution (avoiding deadlock) and the Ethernet interface.

INTERLISP provides two predicates to test for the availability of input from a file: READP and WAITFORINPUT.

14.3.1 READP: Testing Input

READP tests the input buffer of the specified file to determine if any input is available. It returns T if input is available, but NIL otherwise. It takes the form

```
Function:       READP

# Arguments:    2

Arguments:      1) a file name, FILE
                2) a flag, FLG

Value:          T, if there is input waiting in the
                buffer.
```

FLG is used to determine if there is a single EOL (end-of-line) character remaining in the buffer. Normally, READP returns NIL while assuming that the EOL character is left over from the previous input. However, if you want to detect this condition, set FLG to T and READP will return T.

You must exercise caution when using READP. It will return T if there is input in the line buffer for a line that has not been terminated by the user. Even though READP returns T, READ may have to wait until the user terminates the line.

```
←(PROG (char)
       (SETQ char (READC))
       (RETURN (READP)))
WATER⟨CR⟩
T
```

because the characters ATER remain in the input buffer.

```
←(PROG (char)
       (SETQ char (READC))
       (RETURN (READP)))
M⟨CR⟩
T
```

because it sees the ⟨CR⟩ as a character and treats it as remaining input.

```
←(PROG (atm)
       (SETQ atm (READ))
       (RETURN (READP)))
WATER⟨CR⟩
NIL
```

because the ⟨CR⟩ is required to terminate the atom, is read by READ, and so, there is no input remaining in the buffer.

```
←(PROG (atm)
       (SETQ atm (READ))
       (RETURN (READP)))
]⟨CR⟩
T
```

because the] terminates an expression, but the ⟨CR⟩ remains in the input buffer.

If FILE is not open, READP generates an error FILE NOT OPEN ⟨filename⟩.

14.3.2 Waiting for Input

At other times a program may not be able to proceed until it receives input from the terminal or a file. **WAITFORINPUT** allows you to suspend program execution until input is available. It takes the following form:

Function:	WAITFORINPUT
# Arguments:	1
Argument:	1) a file name, FILE
Value:	The name of the file for which input is available.

It waits for input from either the specified file or the terminal. Its value is either T, indicating input is available at the terminal, or the file name, indicating input is available in the file.

WAITFORINPUT operates as a timed wait routine. That is, it waits for a specified amount of time, given by DISMISSMAX, and then returns NIL. This prevents the program from being deadlocked because no input is forthcoming from the terminal or the file.

Internally, WAITFORINPUT uses the algorithm

1. Suspend execution for a time interval, given by DISMISSINIT, the initial interval.
2. Check for input from file or terminal.
3. If input is available, return T or the file name as appropriate.
4. Add the interval to the total wait time.
5. If the total wait time exceeds DISMISSMAX, return NIL.
6. If no input is available, increase the wait interval by a specified factor (1/16 according to the IRM). The wait interval increases up to the maximum interval, given by DISMISSMAX.
7. Suspend execution again.
8. Goto 2.

The value of the argument may also be an integer that specifies how long WAITFORINPUT should wait before returning to the calling function. In this case it waits until the specified number of milliseconds has elapsed and then returns NIL if no input has become available at the terminal. Note that this form overrides the action associated with the value of DISMISSMAX.

WAITFORINPUT will be extremely useful in those environments where you have multiple cooperating processes running within your INTERLISP environment. As processes are only supported under INTERLISP-D, a demonstration of WAITFORINPUT will be deferred until Volume 2.

14.4 CONCEPT OF THE READ TABLE

INTERLISP allows you to decide how characters in the input stream are to be processed. The mechanism for controlling parsing (e.g., interpretation) of input character sequences is driven by a data structure known as the *read table*. Read tables are media-independent. That is, the interpretation of the character is the same regardless of the device from which the read operation is being performed.

Read tables provide the system with information about the syntax class of each character. Characters may assume different roles in different functions or programs depending on their read table descriptions. Roles include specification of breaks, separators, escapes, and list or string delimiters. You may define multiple read tables within a program. You may switch among them by providing the address of the read table as an argument to an input function or you may declare a read table to be effective by a function call.

INTERLISP itself utilizes three default read tables to simplify the interpretation of characters in different subsystems:

T	for terminal input/output
FILERDTBL	for file input/output
EDITRDTBL	for editor input from a terminal

INTERLISP provides functions for changing or resetting the default read tables or copying them to create new specialized ones. You may also create new read tables by specifying their contents explicitly.

14.4.1 Syntax Classes

A *syntax class* is a group of characters that have the same effect with respect to a particular input or output operation. The read table associates a character with its syntax class. When an input or output operation detects a character, it uses the syntax class to determine what to do. There are nine syntax classes in all.

Six of the syntax classes form a special group known as format characters. These classes are

Syntax Class	Initial Value
LEFTBRACKET	[
RIGHTBRACKET	]
LEFTPAREN	(
RIGHTPAREN	)
STRINGDELIM	"
ESCAPE	%

LEFTPAREN and RIGHTPAREN are used to begin and end list structures, respectively.

LEFTBRACKET and RIGHTBRACKET are used to begin and end list structures, respectively. RIGHTBRACKET can be used to close any number of LEFTPAREN lists or back to the last LEFTBRACKET.

STRINGDELIM is used to begin and end a text string. Within a string, all characters except those assigned to the ESCAPE class are treated as ordinary characters (i.e., belonging to class OTHER). A STRINGDELIM character may be included in a text string by prefixing it with an ESCAPE character.

ESCAPE inhibits any special interpretation of the character immediately following it in the character stream (i.e., the character is treated as belonging to class OTHER).

If you assign another character to one of these syntax classes, it does not disable the character that is currently defined. We can make { be interpreted as a LEFTBRACKET character by

```
←(SETSYNTAX '%{ 'LEFTBRACKET)
OTHER

←(SETSYNTAX '%} 'RIGHTBRACKET)
OTHER

←(SETQ x (READ))
{ATLANTA IS A PEACH OF A CITY}
(ATLANTA IS A PEACH OF A CITY)
```

which tells INTERLISP that { is to be treated as a left bracket like [and } as a right bracket like].

Two syntax classes are used to distinguish atoms in an input stream, principally by RATOM. These classes are BREAKCHAR and SEPRCHAR. In fact, there are several break characters recognized by INTERLISP and these are jointly referenced by the syntax class BREAK. BREAK is the union of the classes LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, AND BREAKCHAR.

Both BREAK and BREAKCHAR may be used in SETSYNTAX (see Section 14.4.3). However, you must be careful in using them to define a break character. Compare the results of the following when the input sequence below is to be read:

```
input sequence:      (EQ FRUITS '(APPLES ORANGES))

normal mode:

←(READ)
(EQ FRUITS '(APPLES ORANGES))

alternative mode:

←(SETSYNTAX '%( 'BREAK)
LEFTPAREN
```

```
←(READ)
(EQ fruits '(apples oranges))
(EQ fruits ' (apples oranges))
           ↑
           note the space between ' and the list.
alternative mode:
←(SETSYNTAX '%( 'BREAKCHAR)
LEFTPAREN
←(READ)
(EQ fruits '(apples oranges))
%(
←...)u.b.a
fruits
```

A given character may be disabled by assigning it the syntax class OTHER. Disabling a character means that it no longer is interpreted as a specific class. All characters belonging to the class OTHER have just their literal values when read by INTERLISP.

14.4.2 Getting the Syntax Class

You may get the syntax class of a character by executing **GETSYNTAX**, which takes the form

```
Function:       GETSYNTAX
# Arguments:    2
Arguments:      1) a character, CHAR
                2) a read table, RDTBL
Value:          The syntax class of CHAR.
```

GETSYNTAX returns the syntax class of the character with respect to the specified read table. If read table is T or NIL, the terminal read table is assumed.

```
←(GETSYNTAX '%()
LEFTPAREN
←(GETSYNTAX 'A)
OTHER
←(GETSYNTAX (CHARCODE &))
OTHER
```

```
←(GETSYNTAX 13)
SEPRCHAR
```

You may also specify a syntax class name as the value of the first argument. If so, GETSYNTAX returns a list of the character codes that comprise that class.

```
←(GETSYNTAX 'BREAK)
(34 40 41 91 93 123 125)
←(GETSYNTAX 'SEPRCHAR)
(9 10 12 13 31 32)
```

Special Syntax Functions

Because these expressions are commonly used, INTERLISP provides functions for accessing the break and separator characters of a read table. They take the form

Function:	GETBRK GETSEPR
# Arguments:	1
Arguments:	1) a read table address, RDTBL
Value:	A list of the break or separator characters, respectively, for the specified read table.

Consider the following examples:

```
←(GETBRK)
(34 40 41 91 93)
←(GETSEPR)
(0 9 10 11 12 13 31 32)
←(GETBRK EDITRDTBL)
(23 34 40 41 91 93 124)
```

14.4.3 Setting the Syntax Class

You may set the syntax class using the function **SETSYNTAX**. SETSYNTAX sets the syntax class of a character for a specified read table. The read table may be T or NIL or ORIG which means to set the syntax in the terminal read table. You may also specify either of the other system read tables or one of your own. Its format is

```
Function:       SETSYNTAX

# Arguments:    3

Arguments:      1) a character or character code, CHAR
                2) a syntax class, CLASS
                3) a read table, TABLE

Value:          The previous syntax class of the
                character.
```

If CLASS has the value T or NIL or ORIG, SETSYNTAX assumes that you want to set the syntax class to that defined in the terminal read table:

```
←(SETSYNTAX '%{ 'ORIG)
OTHER
```

means to set the syntax class of { as it originally was. CLASS may also be another read table whence the syntax class of the character in that read table is made the syntax class of the character in the specified read table.

```
←(SETSYNTAX '%{ FILERDTBL EDITRDTBL)
OTHER
```

Finally, CLASS may have the form

```
(⟨type⟩ ⟨options⟩ ⟨function⟩)
```

as described in the definition of read macros (see Section 14.4.5).

Special Syntax Functions

Because users often want to set the break or separator characters to meet specific needs, INTERLISP provides two explicit functions for this operation. They take the form

```
Function:       SETBRK
                SETSEPR

# Arguments:    3

Arguments:      1) a list of character codes, LST
                2) a flag, FLAG
                3) a read table address, RDTBL

Value:          NIL
```

SETBRK and **SETSEPR** set LST as the values of the break and separator characters, respectively, in the specified read table. If LST is T, the separator

characters in RDTBL are set to be those in T, the system read table for terminals, regardless of the value of FLAG.

If FLAG is NIL, RDTBL has only those elements in LST as separator characters; all old separator characters are discarded from RDTBL which are not in LST. If FLAG is 0, it removes from RDTBL as separator characters those characters that appear in LST (e.g., it acts like an UNSETSEPR function). If FLAG is 1, it makes each of the characters in LST be a separator character in RDTBL (e.g., it acts like an OR between LST and RDTBL with respect to separator characters).

If RDTBL is T, then the separator characters are set to be those in the original system table.

The Escape Character

% has a special meaning in INTERLISP to identify non-printing characters. You may cause % to be interpreted like every other character on input using **ESCAPE**, which takes the form

```
Function:       ESCAPE
# Arguments:    2
Arguments:      1) a flag, FLAG
                2) a read table, RDTBL
Value:          The previous setting for the escape
                character.
```

If FLG is NIL, then % is just another character to be read (e.g., it is set to class OTHER). Initially, the setting is T. It returns the previous class of %.

14.4.4 Testing the Syntax Class

You may test the syntax class of a character code within a specific read table using **SYNTAXP**, which takes the form

```
Function:       SYNTAXP
# Arguments:    3
Arguments:      1) a character code, CODE
                2) a syntax class, CLASS
                3) a read table, RDTBL
Value:          T or NIL.
```

SYNTAXP returns T if the character code is a valid member of the syntax class in the specified read table; otherwise, NIL. The syntax class may be any of the standard syntax classes or one of the read macro options that are described in Section 14.4.5. Note that SYNTAXP accepts only character codes.

```
←(SYNTAXP 40 'LEFTPAREN)
T
←(SYNTAXP %( 'LEFTPAREN)
Non-Numeric ARG
(
←(SYNTAXP 100 'OTHER)
T
```

The atom BREAK may be used to refer to the class of all break characters; that is, it is the union of LEFTPAREN, RIGHTPAREN, LEFTBRACKET, RIGHTBRACKET, STRINGDELIM, and BREAKCHAR as explained above. In a similar fashion, the atom SEPR corresponds to all separator characters; it is equivalent to SEPRCHAR.

14.4.5 Read Macros: Defining User Syntax Classes

You may define your own interpretations of characters by associating them with *read macros* in a read table. A read macro is a definition that is substituted for a character when the character is read from an input stream. The definition for a read macro character is given by a function. The interpretation may be modified by one of several options.

Types of Read Macros

There are three types of read macros:

MACRO — When a character is read, its definition is inserted in the expression created by the input function instead of the read macro character. For example, the IRM [irm83] notes that we could define as a read macro as follows:

```
(MACRO
     (LAMBDA (file rdtbl)
          (LIST 'NOT (READ file rdtbl))))
```

so that an expression of the form

```
(AND a b))
```

would be returned as

```
(NOT (AND a b))
```

SPLICE — The definition of the macro is NCONCed into the resulting input

expression. The result of the macro should be a list or NIL. The IRM [irm83] notes that we could define $ as follows:

```
(SPLICE
        (LAMBDA NIL
            (APPEND ⟨atom⟩)))
```

such that if the value of the ⟨atom⟩ is (A B C), then the expression (X $ Y) would be returned as (X A B C Y) when it is read from the input stream.

INFIX — The function that is the definition of the character is called with a list of what has been read. The list is passed to the function in TCONC format as the function's third argument. The function's value is treated as the new TCONC list. The IRM [irm83] notes that + could be defined as follows:

```
(INFIX
        (LAMBDA (file rdtbl dummy)
            (RPLACA (CDR dummy))
                (LIST 'IPLUS
                    (CADR dummy)
                    (READ file rdtbl)))
            dummy))
```

If the function returns NIL, the read macro character is ignored, and reading of characters continues. Otherwise, if the function returns a TCONC list of one element, that element is the value of the READ.

Read macro interpretation may be modified by specifying several options. The options list includes

ALWAYS — The character will always be treated as a break character (except when preceded by an escape character). This is the default interpretation of the macro.

FIRST — The character is not a break character. It will be interpreted as a read macro character only when it is the first character read after a

	break character. For example, ' (single-quote) is interpreted as the abbreviation for (QUOTE ...) only when it is seen as the first character in an expression. Thus, atoms such as DON'T are properly read.
ALONE	The character is not a break character. It is interpreted as a read macro character only if it would have been read as a separate atom. That is, its immediate right and left neighbors are either break or separator characters. For example, * is defined as an ALONE read macro character in order to implement the comment pointer feature.

Note that the three options described above are disjoint; that is, one and only one may be specified for any macro at any instant.

ESCQUOTE ESC	The character will be preceded by an escape character when it is printed by PRIN2. This is the default interpretation.
NOESCQUOTE NOESC	The character is printed without a preceding escape character. For example, ' is a NOESCQUOTE character. The IRM notes that you should be especially careful of declaring NOESCQUOTE read macro characters in FILERDTBL as they are likely not to be read back in properly after they have been written out.

These two options are obviously mutually exclusive. They control whether or not the read macro character is protected by the ESCAPE character on input.

IMMEDIATE	The character is treated as an EOL as soon as it is read. This causes the rest of the line to be passed to the input function. A ⟨CR⟩⟨LF⟩ is printed. These characters take effect

immediately rather than waiting for the line to be terminated normally.

NONIMMEDIATE — The character is not interpreted until a carriage return, matching right parenthesis, or matching right bracket is read. This is the default interpretation.

These two options are obviously mutually exclusive. They determine when a macro's function is actually executed.

The default options are ALWAYS, NONIMMEDIATE, and ESCQUOTE.

Read macros are declared via SETSYNTAX. The form of the declaration is

(⟨type⟩ ⟨options⟩ ⟨function⟩)

14.4.6 Standard Read Macro Characters

INTERLISP defines a number of standard read macro characters: [irm83]:

The single-quote character is defined in the T and EDITRDTBL read tables. It returns the next expression that is read as the argument to the function QUOTE. It is defined as a FIRST read macro so that it has no effect when encountered in the middle of a symbol. It will also be ignored if it is immediately followed by a separator character.

CTRL-Y — This control character is defined in T and EDITRDTBL. Its effect is to return the result of evaluating the next expression that is read. Thus, if you type CTRL-Y followed by an atom or expression, the value of the atom or expression is returned in the result from the input function. For example,

```
←(SETQ keys (LIST ↑Y(ARRAY 10)))
({ARRAYP}#542635)
```

' — The back-quote character acts like quote, except that it causes the expression to be evaluated. The back-quote read-macro acts like a template. An example of its definition is given in Section 14.4.8.

? — This read macro character implements the ?= (on-line help) of the Programmer's Assistant

(see chapter 25). It is defined in T and EDITRDTBL.

`*` This read macro character implements the comment pointer feature. It is defined in FILERDTBL.

CTRL-W This read macro control character is defined in T and EDITRDTBL. In INTERLISP-10, it deletes the entire previous expression that was typed in. In INTERLISP-D it deletes the previously typed "word."

The vertical bar read macro character is equivalent to the back-quote read macro since some terminals do not have the back-quote symbol on their keyboard. It is also used to support the printing and reading of unusual data structures (see HPRINT).

14.4.7 Read Macro Functions

Read macro functions allow you to enable or disable the effect of read macros, and to test whether or not any read macros are in effect.

Enabling or Disabling Read Macros

READMACROS enables or disables the effect of any read macros that are defined. It takes the form

```
Function:       READMACROS
# Arguments:    2
Arguments:      1) a flag, FLAG
                2) a read table, RDTBL
Value:          The previous setting.
```

READMACRO is used to enable or disable the effect of read macros in the specified read table. If FLAG is NIL, the effect of read macros is disabled. A value of T for FLAG will enable the effect of read macros.

Testing for Read Macro Execution

You may test if your program is executing under a read macro function using **INREADMACROP**, which takes the form

```
Function:       INREADMACROP
# Arguments:    0
```

Arguments:	NIL
Value:	The number of unmatched parentheses or brackets; otherwise, NIL.

INREADMACROP returns NIL if you are not currently executing under a read macro function, Otherwise, it returns the number of unmatched left parentheses or left brackets.

Setting the Read Macro Flag

SETREADMACROFLG sets the internal system flag that informs READ that it is under a read macro function. It takes the following form:

Function:	SETREADMACROFLG
# Arguments:	1
Argument:	1) a flag value, FLG
Value:	The old value of the flag.

When the flag is set, unmatched right parentheses or brackets will cause the error message "READ MACRO CONTEXT ERROR" to be displayed and a break to occur. This function disables that error message when debugging read macro functions.

Note that the read macro functions are primarily for use in functions that are associated with read macros.

14.4.8 BQUOTE: An Example of a SPLICE Macro

BQUOTE is a *back-quote* facility provided as a Lisp User package. It takes forms having the structure '⟨form⟩ and treats them as '⟨form⟩ except that any expression preceded by a "," will be evaluated. This facility is similar to a read macro provided by MACLISP [char80]. BQUOTE handles several different forms:

1. An expression preceded by a ",' is merely evaluated and the value substituted in the form for the expression.
2. An expression preceded by ",@' is evaluated and spliced in using APPEND.
3. An expression preceded by ",." is spliced in using NCONC.

Consider the following example:

```
'(A ,B ,@C ,.D E)
```

is equivalent to

```
(CONS 'A
      (CONS B
            (APPEND C
                    (NCONC D '(E)))))
```

BQUOTE is implemented as a read macro. When '⟨form⟩ is read, it is transformed to (BQUOTE ⟨form⟩) and evaluated. Loading the BQUOTE package places the read macro in T, FILERDTBL, and EDITRDTBL.

A Definition for BQUOTE

BQUOTE is declared as follows:

```
(SETSYNTAX '
           '(SPLICE
              (LAMBDA (file rdtbl)
                   (SELECTQ (PEEKC file)
                        (' (READC file)
                           (LIST (READBQUOTE file
                           rdtbl)))
                        NIL)))
           FILERDTBL)
```

READBQUOTE is defined as follows:

```
(DEFINEQ
 (readbquote (file rdtbl)
(RESETLST
   (RESETSAVE
      NIL
      (LIST 'SETSYNTAX
            ',
            (SETSYNTAX
               ',
               '(MACRO FIRST
                     (LAMBDA (file rdtbl)
                             (SETQ comma
                                   (SELECTQ (PEEKC
                                   file)
                                       (%. (READC file)
                                           ',.)
                                       (, (READC file)
                                           ',,)
```

```
                                                    (! (READC file)
                                                       ',!)
                                                    (@ (READC file)
                                                       ',@)
                                                    ',))))
                    rdtbl)
                  rdtbl))
    (PROG (comma form)
          (SETQ form (READ file rdtbl))
          (RETURN
                (COND
                     (comma (LIST 'BQUOTE form))
                     (T
                          (KWOTE form))))))
))
```

14.5 READ TABLE FUNCTIONS

A number of functions are provided to allow you to manipulate read tables. Note that you may have any number of read tables defined in your applications environment, but only one may be active for each of the three functional areas: T, EDITRDTBL, and FILERDTBL. However, you may replace the system default read tables with ones of your own choosing, and you may do so on a dynamic basis.

14.5.1 Testing a Read Table

READTABLEP allows you to test whether or not an object is the address of a read table. It takes the form

Function:	READTABLEP
# Arguments:	1
Argument:	1) a read table address, RDTBL
Value:	RDTBL, if RDTBL is the address of a read table.

Note that READTABLEP returns T if the terminal read table or system read table (i.e., ORIG) happens to be the object specified.

14.5.2 Obtaining a Read Table Address

INTERLISP maintains several different read tables which are selected depending on the input function that is to be performed. You may obtain the address of a specific read table using **GETREADTABLE**, which takes the following form:

```
Function:       GETREADTABLE
# Arguments:    1
Arguments:      1) a read table address, RDTBL
Value:          The address of a read table.
```

If RDTBL is the address of a valid read table, this value is returned as the result of GETREADTABLE. Usually, RDTBL takes the values T or NIL which are interpreted as follows:

1. If RDTBL is T, GETREADTABLE returns the address of INTERLISP's read table for terminals.
2. If RDTBL is NIL, GETREADTABLE returns the address of the primary read table.

If RDTBL is not the address of a read table, INTERLISP generates an error message "ILLEGAL READTABLE".

INTERLISP maintains a separate read table for reading data from files. The address of this read table is stored in the variable FILERDTBL.

14.5.3 Setting a Read Table

You may set the primary system read table or the terminal read table using **SETREADTABLE**, which takes the form

```
Function:       SETREADTABLE
# Arguments:    2
Arguments:      1) a read table address, RDTBL
                2) a flag, FLG
Value:          The previous address of the read table.
```

SETREADTABLE sets the primary read table to the read table whose address is given as the value of RDTBL. In addition, if FLG is T, SETREADTABLE also sets the terminal read table as well.

You may reset the other read tables by assigning the new read table address to the appropriate variable: EDITRDTBL for the editor or FILERDTBL for the File Package.

14.5.4 Copying a Read Table

COPYREADTABLE returns the address of a copy of the specified read table. It takes the form

```
Function:       COPYREADTABLE
# Arguments:    1
Argument:       1) a read table address, RDTBL
Value:          The address of a new read table.
```

COPYREADTABLE is the only way by which you can create a new read table in INTERLISP. RDTBL is interpreted as follows:

1. If RDTBL is the address of a valid read table, a copy of that read table is created and its address is returned to you.

2. If RDTBL is NIL, a copy of the primary system read table is made and its address is returned to you.

```
←(COPYREADTABLE)
[READTABLEP]#542224
```

3. If RDTBL is T, a copy of the system terminal read table is made and its address is returned to you.

```
←(COPYREADTABLE T)
[READTABLEP]#542427
```

4. If RDTBL is ORIG, a copy of the original system read table is made and returned to you.

```
←(COPYREADTABLE 'ORIG)
[READTABLEP]#542632
```

5. Otherwise, an error is generated: ILLEGAL READTABLE.

14.6 LINE BUFFERING

Characters typed in at your terminal are normally placed in a line buffer. When an input function requests a character or sequence of characters, they are extracted from the line buffer and transferred to the requesting function when a carriage return is typed. Until a carriage return is typed, you may delete characters from the line buffer by typing CTRL-As or delete its entire contents by typing CTRL-Q.

An important consideration in line buffering is which function will process the input. This function determines whether or not parenthesis counting is observed.

```
←(PROGN
     (RATOM)
     (READ))
(rigel betelgeuse polaris)
```

where you typed in

```
aldebaran (rigel betelgeuse polaris)⟨CR⟩
```

This expression requires a carriage return at the end of the line before any action is taken. The first function, RATOM, requires a carriage return before it receives its input even though that input is already in the buffer.

Alternatively, if you typed in

```
←(PROGN
     (READ)
     (READ))
(rigel betelgeuse polaris)
```

with the same input sequence except for the ⟨CR⟩, no carriage return is required because READ will accept the next atom (see Section 14.1).

INTERLISP provides you with a substantial amount of control over line buffering. Many application programs need to get their data in certain sequences or ensure that all of the data are entered before they begin to process it. This section describes how to control line buffering.

Note that the following functions discuss terminal tables, which are described in more detail in Section 15.5.

14.6.1 Enabling and Disabling Line Buffering

You may enable or disable line buffering by executing **CONTROL**, which takes the form

```
Function:       CONTROL
# Arguments:    2
Arguments:      1) a control mode, MODE
                2) a terminal table address, TTBL
Value:          The previous control setting.
```

If MODE is T, line buffering is disabled for the specified terminal table. Line buffering will not be effected until TTBL is made the current terminal table via SETTERMTABLE (unless it already is the current terminal table—when TTBL is NIL). Normally, CONTROL is invoked with TTBL equal to NIL.

The current control mode may be obtained by executing **GETCONTROL** with a terminal table address, TTBL. GETCONTROL takes the following form:

```
Function:       GETCONTROL
# Arguments:    1
Argument:       1) a terminal table address, TTBL
Value:          The current control mode for TTBL.
```

Consider the following example,

```
←(GETCONTROL)
NIL
←(SETQ NEW.TTBL (COPYTERMTABLE))
[TERMTABLEP]#1,104640
←(CONTROL T NEW.TTBL)
NIL
←(GETCONTROL NEW.TTBL)
T
```

When line buffering is disabled, the function initiating input determines how the typed-in characters are treated when (CONTROL T) is in effect. There are several cases:

READ

If the input sequence is a list (determined by a leading "(" or "["), then the input is buffered until a carriage return or matching closing parenthesis or bracket is detected. That is, the effect is the same as if CONTROL is NIL.

If the input sequence is not a list, a break character or a separator character terminates the character sequence and causes it to be transferred to the requestor. CTRL-A and CTRL-Q are both operable, but only on the most recent atom.

```
(DEFINEQ
     (read.without.buffering (ttbl)
          (PROG (line chars)
                (CONTROL T ttbl)
                (SETQ line NIL)
          loop
                (SETQ chars (READ))
                (COND
                    ((NULL chars)
                         (PROGN
                              (CONTROL NIL ttbl)
                              (RETURN line)))
```

```
                ((NULL line)
                    (SETQ line (LIST chars)))
                (T
                    (SETQ line
                        (APPEND line (LIST
                        chars)))))
            (GO loop))
    ))
```

Now, let us execute this function:

```
←(READ.WITHOUT.BUFFERING NIL)
Every goode good boy deserves flavour favor ⟨CR⟩
          ↑                              ↑
      CTRL-Q                         CTRL-Q
returns: Every good boy deserves favor
```

Note that CTRL-Q only deletes the last character sequence that was entered because each sequence separated by a blank is transmitted to READ.

RATOM

A sequence of characters is returned as soon as a break or separator character is typed in. Until that time, CTRL-A and CTRL-Q have their usual effect.

```
←(CONTROL T)
NIL
←(RATOM)
(⟨CTRL-A⟩##
```

returns (because (is a break character. It would already have been transmitted. ## indicates that CTRL-A was typed in while the buffer was empty.

READC/PEEKC

A character is transmitted immediately. No line editing is possible.

```
←(CONTROL T)
NIL
←(READC)⟨CTRL-A⟩
⟨space⟩
```

because CTRL-A forces READC to return NIL, so the result is the null character code.

14.6.2 Clearing the Line Buffer

CLEARBUF clears the line buffer for a file from which a program is reading. It takes the form

```
Function:       CLEARBUF
# Arguments:    2
Arguments:      1) a file name, FILE
                2) a flag, FLAG
Value:          NIL
```

Input is buffered for every from file from which INTERLISP can read. Usually, a program reading from the terminal discovers that the user has entered some erroneous expression. At that point, rather than process the expression, it clears the line buffer and prints a warning message or otherwise handles the error. To clear the terminal line buffer, the program merely executes

```
←(CLEARBUF T T)
NIL
```

This form of CLEARBUF saves both the line buffer and the system buffer for later access by the program. However, if both the line buffer and the system buffer are empty, nothing is stored into the internal save buffers.

Certain control characters also save the line and system buffers. CTRL-D, CTRL-E, CTRL-H, CTRL-P, and CTRL-S cause INTERLISP to automatically perform (CLEARBUG T T). After the user has finished his interaction with CTRL-P or CTRL-S, the line and system buffers are restored.

14.6.3 Accessing the Buffer's Contents

After the line and system buffers are saved, you may access their contents by executing either **LINBUF** or **SYSBUF**. They take the form

```
Function:       LINBUF
                SYSBUF
# Arguments:    1
Arguments:      1) a flag, FLAG
Value:          The contents of the internal save buffer.
```

If FLAG is T, the contents of the line buffer or the system buffer are returned to the user from the internal save buffers. If FLAG is NIL, the respective internal buffer will be cleared.

14.6.4 Resetting the Line and System Buffers

After you have cleared the line and system buffers, you may reset them to their original contents or some modified version of their contents before processing the input. Two functions, **BKLINBUF** and **BKSYSBUF** set the line and system buffers, respectively, to the values of their arguments. BKSYSBUF takes the form

```
Function:       BKSYSBUF
# Arguments:    3
Argument:       1) a string, X
                2) a flag, FLAG
                3) a read table, RDTBL
Value:          The PRIN1-name of X.
```

BKSYSBUF set the buffer to the PRIN1-name of X. If the length of the string is greater than 160 characters, only the first 160 characters will be restored.

Consider the following examples:

```
←(BKSYSBUF "(ADD1 1)")
"(ADD1 1)"
←(ADD1 1)
2
←(BKLINBUF "I love New York")
"I love New York"
←(BKSYSBUF "(SETQ x (READ))")
"(SETQ x (READ))"
```

which causes the following to appear in the system buffer to be evaluated:

```
←(SETQ x (READ))
↑
cursor waits for you to enter something, for example
(I love New York)
(I love New York)
```

BKLINBUF takes the form

```
Function:       BKLINBUF
# Arguments:    1
```

```
Argument:       1) a string, STRING
Value:          The value of STRING.
```

BKLINBUF sets the INTERLISP line buffer to the value of STRING. The length of the line buffer may be limited in certain implementations by the host computer operating system, in which case the extra characters are ignored.

14.7 THE ASKUSER PACKAGE

Many programs engage in an interactive dialogue with a user before, during, and after execution of their primary functions. To facilitate the construction of dialogues and interpretation of the responses, INTERLISP provides the Askuser Package.

Askuser is driven by a *key list*, a list of expressions that describe

what may be entered by the user,
what responses should be given to the user,
what result should be returned to the calling function.

14.7.1 ASKUSER

ASKUSER allows your program to conduct an interactive dialogue with a user to acquire information about what it should do next. It takes the form

```
Function:       ASKUSER
# Arguments:    8
Arguments:      1) a time period, WAIT
                2) a default answer, DEFAULT
                3) a message, MESSAGE
                4) a key list, KEYLST
                5) a type ahead flag, TYPEAHEAD
                6) a LISPX flag, LISPXPRINTFLAG
                7) an options list, OPTIONSLST
                8) a file name, FILE
Value:          As described below.
```

ASKUSER displays MESSAGE at the terminal (unless FILE is non-NIL). It then issues a READC to get a character from type in. As each character is typed, it is matched against the keys in KEYLST. If a match is successful, the character (or its replacement) is echoed or a new prompt is displayed as determined by the options. If matching is unsuccessful, the character is deemed unacceptable (un-

less it appears in a MACROCHARS entry) and ASKUSER rings the bell or otherwise signals you.

```
←(ASKUSER 100 '(N NO) "DO YOU WANT TO CONTINUE?" '(Y N))
N
N          "returned by ASKUSER"
```

At any time, you may type ?. ASKUSER will display a list of acceptable responses either drawn from the key list entries or as determined by EXPLAINSTRING if it appears in OPTIONSLST. You may also type one of several control characters (CTRL-A, CTRL-Q, CTRL-X) which will cause ASKUSER to reinitialize and begin anew.

```
←(ASKUSER NIL
          '(RED)
          "WHAT IS YOUR FAVORITE COLOR?"
          '(RED BLUE ORANGE GREEN YELLOW PURPLE))
?
one of:
RED
BLUE
ORANGE
GREEN
YELLOW
PURPLE
```

Keys may be atoms or strings. Thus, ASKUSER will not return a value until you have entered an acceptable response.

ASKUSER waits for WAIT seconds after printing MESSAGE. If WAIT is NIL, ASKUSER will wait forever. If the waiting period expires with no user type in, ASKUSER supplies the default response given by DEFAULT after printing "...". DEFAULT must be one of the keys in KEYLST. It is treated exactly as if it had been entered from the keyboard. Typically, N or NO is used as the value of DEFAULT.

KEYLST is a key list (as described in Section 14.7.2). OPTIONSLST allows you to define a global set of options for ASKUSER rather than associating options with each key entry.

If you want to type ahead, then TYPEAHEAD should be T. Otherwise, characters typed in ahead of ASKUSER are cleared and saved.

LISPXPRINTFLAG determines whether or not the interaction is recorded on the history list (see Chapter 27).

If FILE is non-NIL, characters are read from the file until an unacceptable input is encountered or a key is complete. FILE may be either a file name or a string. If it is a string, and all its characters are read without completing a key, FILE is reset to T and interaction continues with the user via type in.

14.7.2 Key Completion

ASKUSER attempts to match characters that it reads until it completes a key. Deciding if a key is complete is more complex than whether or not all of its characters have been matched. There may be multiple keys with characters in corresponding positions that would match the most recently read character. ASKUSER would not know which key to match without reading additional characters.

A key is considered complete after N characters have been read if

1. All of its characters have been matched and no other key has the key as a substring.
2. All of its characters have been matched and you have entered a confirmation character. The confirmation character serves both to complete the key and confirm it even when CONFIRMFLG is NIL.

This allows you to complete a key through confirmation when the key is a substring of another key, even when CONFIRMFLG is NIL. The confirming character serves to both complete the key and confirm it.

3. All of its characters have been matched, and CONFIRMFLG is NIL, and the KEYLST option has a value, then if the next characters matches one of the keys in the value of the KEYLST option.

   The IRM suggests the following example:

```
(ST "ore and redefine "
        CONFIRMFLG T
        KEYLST ("" (F . "orget exprs"))
```

   Suppose there was another key that was STX. If you had typed ST, then there are still two ways to complete the key. Typing F completes the ST key because it matches the F of the inner key list.
4. There is only one key left and you have typed a confirming character. If CONFIRMFLG is T, you must still confirm a key whether or not it is complete. Confirmation allows you to complete a key even though you have not entered all of the required characters.

14.7.3 Key List Format

A key list is a list of expressions that drive the actions of ASKUSER. A key list takes the form

```
(⟨key⟩ ⟨promptstring⟩ ⟨options⟩)
where:    ⟨key⟩                is an atom or a string.
```

⟨promptstring⟩	is an atom or string that is printed when the key is recognized.
⟨options⟩	is a list of options.

Options may be specified in two ways:

1. If an option appears in ⟨options⟩, its value is the next element in the list. That is, ⟨options⟩ has a property list format.
2. If the option appears in OPTIONSLST, its value is the next element in OPTIONSLST. OPTIONSLST serves as a default for the entire key list which may have several levels of structure.

If an option does not appear on ⟨options⟩ or OPTIONSLST, its value is assumed NIL for this invocation of ASKUSERS.

14.7.4 The Default Key List

If ASKUSER is called with KEYLST having the value NIL, it uses a default key list. The structure of this key list is

```
((Y "es⟨cr⟩") (N "o⟨cr⟩"))
```

```
←(ASKUSER NIL 'N "Type ? for Responses")
Type ? for Responses
←?
Yes
No
←
```

All other inputs are unacceptable. ASKUSER signals this fact by ringing the bell (or sounding some other terminal chime). It does not echo the erroneous character.

14.7.5 Key List Options

The following options may occur in the key list ⟨options⟩ segment or in OPTIONSLST:

CONFIRMFLG	If this flag is T, the key must be confirmed by typing a ⟨cr⟩ or a space after the last character. If its value is a list, the confirmation character

```
                                may be any member of the list
                                (in addition to ⟨cr⟩ and
                                ⟨space⟩).

←(ASKUSER NIL
          'N
          "Do You Like Ice Cream?"
          '(( Y "es") (N "o") (M "aybe"))
          NIL
          NIL
          '(CONFIRMFLG T))
Do You Like Ice Cream?
Yes⟨CR⟩
Y
```

where the "es" was printed by ASKUSER. Then, I had to terminate the input with a ⟨CR⟩ because the CONFIRMFLG had the value T.

```
PROMPTCONFIRMFLG        If T, whenever confirmation is
                        required, you will be prompted
                        with the message "[confirm]".
```

For example, let us modify the options list above to add (PROMPTCONFIRMFLG T):

```
←(ASKUSER NIL
          'N
          "Do You Like Ice Cream?"
          '(( Y "es") (N "o") (M "aybe"))
          NIL
          NIL
          '(CONFIRMFLG T PROMPTCONFIRMFLG T))
Do You Like Ice Cream?
Yes [confirm] ⟨CR⟩
Y
```

```
PROMPTON                If non-NIL, the PROMPTSTRING
                        will be printed only when the
                        key is confirmed with a member
                        of its value.

COMPLETEON              When a confirming character is
                        read, the value of this option
                        is used to automatically
                        complete the key. These
                        characters are printed only when
                        the key is confirmed by a member
                        of PROMPTON.
```

Let us modify the example above to demonstrate these features:

```
←(ASKUSER NIL
          'N
          "Do You Like Ice Cream?"
          '(( Y "es, if you really like it.")
            (N "o, if you don't like it.")
            (M "aybe, if you are not sure!"))
          NIL
          NIL
          '(CONFIRMFLG ($) PROMPTON ($) COMPLETEON ($)))
Do You Like Ice Cream?
Y$es, if you really like it.
Y
```

where the $ is entered by the user, and the remaining characters are used to complete the answer.

NOCASEFLG	If T, case-independent matching is not performed; otherwise, characters typed in must match the case of characters in the key.
NOECHOFLG	If non-NIL, matched characters are not echoed. Also, the confirming character will not be echoed.
AUTOCOMPLETEFLG	If non-NIL, ASKUSER attempts to complete the key after each character is typed rather than waiting for the entire key to be entered. ASKUSER will only attempt to complete an unambiguous key.

```
←(ASKUSER NIL
          'N
          "Do You Like Ice Cream?"
          '(( YES ", if you really like it.")
            (NO ", if you don't like it.")
            (MAYB ", if you are not sure!"))
          NIL
          NIL
          '(CONFIRMFLG ($) AUTOCOMPLETEFLG T))
Do You Like Ice Cream?
YES, if you really like it.⟨CR⟩
YES
```

Note that the key was completed automatically once I typed the Y via ASKUSER printing "ES, if you really like it." and then waiting for me to terminate input.

RETURN — If non-NIL, its value is evaluated and returned as the value of executing ASKUSER. Different keys may have different return values.

```
←(ASKUSER NIL
          'N
          "Do You Like Ice Cream?"
          '(( YES ", if you really like it.")
            (NO ", if you don't like it.")
            (MAYB ", if you are not sure!"))
          NIL
          NIL
          '(CONFIRMFLG T
            AUTOCOMPLETEFLG T
            RETURN 'CHOCOLATE))
Do You Like Ice Cream?
YES, if you really like it.⟨CR⟩
CHOCOLATE
```

EXPLAINSTRING — If non-NIL, its value is printed when a user types a ? while ASKUSER expects a key. Normally, ASKUSER will display the possible key values in response to a ?. Using EXPLAINSTRING, you may provide detailed explanations of what is expected to the user.

KEYSTRING — If non-NIL, matched characters in the key are echoed by the corresponding characters in KEYSTRING. Its primary use is for echoing lower-case characters.

MACROCHARS — Its value is a list of detailed pairs of the form

(⟨character⟩ . ⟨expression⟩)

	If a character that has been typed in does not match any of the keys, but does occur in a MACROCHAR entry, the expression is evaluated. This implements a read macro capability inside ASKUSER.
EXPLAINDELIMITER	If non-NIL, its value is printed to separate the explanation from ?. Initially, its value is ⟨cr⟩. You may set it to any character or combination of characters represented as a string.

14.7.6 Key List Construction

Key list construction can be simplified when you do not want to provide prompt strings or options for key entries. **MAKEKEYLST** constructs a simple key list for you. It takes the form

Function:	MAKEKEYLST
# Arguments:	3
Arguments:	1) a list of keys, LST 2) a default key, DEFAULTKEY 3) a lower-case flag, LCASEFLG
Value:	The selected key.

MAKEKEYLST takes a list of atoms or strings, LST, and constructs a key list. DEFAULTKEY, if non NIL, is the last key in the key list. It permits you to specify "none of the above" conditions. Otherwise, a key which permits you to specify "No" is included in the key list. If LCASEFLG is T, keys represented as upper case in the key list will be echoed in lower case.

You may specify a selection by typing the characters of a key or a number corresponding to its position in the key list. Consider the following example:

```
←(MAKEKEYLST '(CONNECT PRINT LOAD MAKE) 'NO T)
((CONNECT NIL
          KEYSTRING "CONNECT"
          CONFIRMFLG T
          AUTOCOMPLETEFLG NIL
          RETURN 'CONNECT
 (PRINT   NIL
          KEYSTRING "PRINT"
```

```
            CONFIRMFLG T
            AUTOCOMPLETEFLG NIL
            RETURN 'LOAD)
   (LOAD    NIL
            KEYSTRING "LOAD"
            CONFIRMFLG T
            AUTOCOMPLETEFLG NIL
            RETURN 'LOAD)
   (MAKE    NIL
            KEYSTRING "MAKE"
            CONFIRMFLG T
            AUTOCOMPLETEFLG NIL
            RETURN 'MAKE)
   (1 "CONNECT" NOECHOFLG T
                EXPLAINSTRING "1-CONNECT"
                CONFIRMFLG T
                RETURN (PROGN
                          (TERPRI T)
                          'CONNECT))
   ...
   ("No-none of the above" " "
                            CONFIRMFLG T
                            AUTCOMPLETEFLG T
                            RETURN NIL))
```

14.7.7 Special Keys

ASKUSER allows you to use a few symbols to have special meaning in the key list. These include

`&` This symbol can be used as a key that will match any single character, provided the character does not match with some other key at that level.

`$` This symbol can be used as a key to match with the result of a single invocation of READ. $ allows you to have a READ that is errorset protected within ASKUSER, so that CTRL-E will allow you to erase the last character and continue with another key.

`$$` This symbol can be used as a key to match a single invocation of READLINE.

`⟨list⟩` The ⟨list⟩ is evaluated as an expression. Its value is used to match the key. This allows you

to provide for arbitrary input operations for any ASKUSER request. That is, besides the standard keys, you may also allow a user to enter other commands that are outside the purview of the current sequence.

" " The empty string can be used as a key. Because it has no characters, all of its characters are automatically matched. It usually is used as a place marker. The IRM notes that one of the entries on the key list for ADDTOFILES? appears as

```
(" " "File/List: "
    EXPLAINSTRING "a file name or name of a function
    list"
    KEYLST ($))
```

Thus, if you type a character that does not match any of the other characters in the key list, the character completes the " " key because it matches with the $ in the inner key list.

15

Output Functions

The corollary to entering data into a program is to emit data from it. There are two aspects to this ability:

1. To present data, possibly formatted, in such a way that it is both readable and understandable by the program user.
2. To preserve data on an external file system such that its lifetime exceeds the execution of the program.

This chapter discusses the output functions provided by INTERLISP. Most functions work equally well whether the output is directed to the terminal (default) or to an external file (specified by an optional argument). Exceptions will be noted in the text.

Many of the functions discussed below will take an optional argument which is a file name. If this argument is not present, output is directed to the terminal. If it is present and the file is open for output, the output will be directed to the file. Some of the functions also take a read table as an argument. This will ensure that data written out may be read in properly under control of the same read table.

INTERLISP will print out the contents of standard objects such as atoms, lists, strings, and numbers. Other objects are printed in the form

```
[⟨datatype⟩]#⟨address⟩
```

where ⟨datatype⟩ is the type of object, such as ARRAY, and ⟨address⟩ is its location in memory. To display the contents of these objects, you must use DEFPRINT to specify functions that know how to print specific objects.

15.1 PRINTING S-EXPRESSIONS: PRINX

INTERLISP programs manipulate S-expressions be they atoms or lists. The basic output facility provides for the display of S-expressions. Indeed, most of the display from your programs will be in the form of S-expressions. Three functions can support the bulk of your printing requirements. The effect of some of these functions may be mediated by a read table.

15.1.1 PRIN1

PRIN1 is a function that prints the value of its first argument. An optional second argument may specify an external file. Its format is

```
Function:       PRIN1
                PRIN3

# Arguments:    1-2

Arguments:      1) an S-expression, EXPRESSION
                2) a file name (optional), FILE

Value:          The value of the S-expression.
```

PRIN1 prints the value of EXPRESSION on the specified file. If file is not specified, the primary output file (see Section 16.2) will be used. In most cases, this will be your terminal.

```
←(SETQ fruit 'orange)
orange

←(PRIN1 fruit)
orange

←(PRIN1 ''database engineering")
database engineering
```

Note that PRIN1 prints the contents of the string without its enclosing double-quotes.

```
←(SETQ presidents
       (APPEND (LIST 'kennedy 'johnson)
               (LIST 'nixon 'ford)))
(kennedy johnson nixon ford)

←(PRIN1 presidents)
(kennedy johnson nixon ford)
```

PRIN1's actions are not mediated by a read table. Thus, it is primarily used for printing formatting characters.

```
←(PRIN1 '%[)
[          enables us to print a left square bracket
←(PRIN1 100)
100
←(RADIX 8)
12
←(PRIN1 100)
144
```

Note the change in radix affects the resulting output.

If the value of the argument is something other than an atom, number, list, or string, PRIN1 will print the value of the pointer to the INTERLISP object. The pointer describes the storage location of the object.

```
←(PRIN1 (ARRAY 10))
{ARRAYP}#542224
```

PRIN3 is a variant of PRIN1 that does not increment the position counter for the output line. It may be used to place control characters in the output buffer. No line length checks are performed when PRIN3 is used.

15.1.2 Printing with Separators

PRIN2 prints S-expressions on a file such that they may be properly read back into the program by READ using a read table. The format for PRIN2 is

```
Function:       PRIN2
                PRIN4
                SHOWPRIN2

# Arguments:    1-3

Arguments:      1) an S-expression, EXPRESSION
                2) a file name (optional), FILE
                3) a read table (optional), RDTBL

Value:          The value of the S-expression.
```

PRIN2 prints all break and separator characters in the S-expression. Contrast the printing of strings by PRIN2 with that of PRIN1:

```
←(PRIN2 "database engineering")
"database engineering"
```

where " is a break character that denotes a string.

```
←(PRIN2 '%[)
%[              where % is the escape character.
←(PRIN2 100)
100
←(RADIX 8)
12
←(PRIN2 100)
144Q
```

Note that PRIN2 inserts a Q after the number when the radix has been changed while PRIN1 does not. Both functions will print the number in octal.

PRIN2 is used for printing S-expressions to a file that may be read back into INTERLISP via READ. It is used by most of the File Package functions for printing various types of S-expressions to a file. The break and separator characters are preceded by % (the escape character) so that they are read back in properly.

If the value of the argument is something other than an atom, number, list, or string, PRIN2 will print the value of the pointer to the INTERLISP object exactly like PRIN1. The pointer describes the datatype and storage location of the object.

PRIN4 is a variant of PRIN2 that does not increment the position counter for the output line. It may be used to place control characters in the output buffer. No line length checks are performed when PRIN4 is used.

SHOWPRIN2 acts like PRIN2 except that its prettyprints the value of the S-expression. Prettyprinting (see Section 15.7) is enabled by setting SYSPRETTYFLG to T. Initially, the value of SYSPRETTYFLG is NIL.

A Definition of SHOWPRINT

We might define SHOWPRIN2 as follows:

```
(DEFINEQ
   (showprin2 (expression file rdtbl)
      (COND
         (SYSPRETTYFLG
            (*
                  If the system prettyprint flag is
                  set, the value of the flag may be the
                  function to be used to perform the
                  prettyprinting operation.

                  Otherwise, we just use PRINTDEF,
                  which is the standard prettyprinting
                  function.
```

```
            (RESETFORM (OUTPUT FILE)
                       (APPLY*
                            (COND
                                 ((GETD SYSPRETTYFLG)
                                      SYSPRETTYFLG)
                                 (T 'PRINTDEF))
                            expression
                            T)))
       (T
            (*
                 Otherwise, just use PRIN2 to do the
                 job as the user intended.
            )
            (PRIN2 expression file rdtbl)))
       expression
))
```

15.1.3 Printing with a Carriage Return

Neither PRIN1 nor PRIN2 terminates the output line after printing the value of the S-expression. To do so, you must explicitly execute (TERPRI) to place a carriage return in the output buffer. However, using **PRINT**, you may combine the effects of both PRIN2 and TERPRI in one function. PRINT takes the format

```
Function:       PRINT
                SHOWPRINT

# Arguments:    1-3

Arguments:      1) an S-expression, EXPRESSION
                2) a file name (optional), FILE
                3) a read table (optional), RDTBL

Value:          The value of the S-expression.
```

PRINT uses PRIN2 to display the value of the S-expression and then places a carriage return in the output buffer.

```
←(PRIN2 (ARRAY 10))
{ARRAYP}#542254{ARRAYP}#542254

←(PRINT (ARRAY 10))
{ARRAYP}#542240
{ARRAYP}#542240
```

In the first example, PRIN2 does not insert a ⟨CR⟩, so the value printed and the value returned by the function appear on the same line. In the second exam-

ple, PRINT has placed a ⟨CR⟩ after the object pointer. Note that INTERLISP automatically inserts a line feed after a carriage return in the output buffer.

SHOWPRINT acts like PRINT except that it prettyprints the value of the S-expression. Prettyprinting (see Section 15.7) is enabled if SYSPRETTYFLG has the value T. Initially, SYSPRETTYFLG is NIL.

15.1.4 Printing Bells

Most users now interact with computer systems via video display units (VDUs) which are, for the most part, silent. Sometimes, the system or a program that you have written will want to catch your attention. It can do so by printing a sequence of bells via **PRINTBELLS**, which takes the form

```
Function:       PRINTBELLS
# Arguments:    0
Argument:       NIL
Value:          " " (e.g., a string with non-printing bell
                character codes).
```

PRINTBELLS is an anachronism remaining from the early days of computing when most interactive usage was performed with Teletypes or similar devices. These devices had a physical bell which rang when you reached the end of the carriage on the printer. The bell could be sounded by a program by sending the appropriate character code to the device (e.g., hexadecimal 2F for an IBM System/370 machine). Today, most VDUs have an audible signal (not necessarily a bell) which performs the same function. PRINTBELLS sends a sequence of character codes to the device that result in the audible signal being produced by the device.

15.1.5 User Defined Printing

As noted above, the printing routines will only print the values of LISP objects which are atoms, numbers, strings, or lists. Other LISP objects are normally represented by pointers to their storage locations. You may provide your own printing routines to handle other types of LISP objects via **DEFPRINT**, which takes the form

```
Function:       DEFPRINT
# Arguments:    2
Arguments:      1) an INTERLISP object type, TYPE
                2) a function, FN
Value:          T
```

TYPE is either a type name, such as ARRAYP, TERMTABLEP, etc., or a type number.

DEFPRINT specifies a function that will handle the printing of LISP objects of the specified type. Whenever a LISP object of TYPE is encountered by one of the printing routines, it calls FN with the LISP object as its argument. The function must be defined in the normal manner using DEFINE or DEFINEQ.

FN may do one of three things:

1. It may return NIL, whence the object will be printed using the system default.
2. It may print the object according to your specifications in the code. It should return (NIL) to prevent any further action by the calling routine.
3. It may return a specification for how the object is to be printed. This specification takes the form

```
(⟨item1⟩ . ⟨item2⟩)
```

ITEM1 will be printed using PRIN1 unless it is NIL, whence no action will occur. ITEM2 is then printed using PRIN2 with no intervening spaces between the two items. The IRM notes that ITEM1 will typically be a READMACRO character.

Consider a function for printing an array of numbers. We would like to print the elements of an array as a list each of whose elements is a CONS cell. The CAR of the CONS cell is the index and the CDR is the element at that location. Let us define PRINT.ARRAY to perform this operation.

```
←(DEFINEQ
    (print.array (x)
        (PROG (size)
            (COND
                ((NOT (ARRAYP x))
                    (ERROR "Not an array" x)))
            (SETQ size (ARRAYSIZE x))
            (FOR I FROM 1 to SIZE
                DO
                    (PRINT (CONS I (ELT x I)))))
    ))
(PRINT.ARRAY)

←(DEFPRINT 'ARRAYP (FUNCTION PRINT.ARRAY))
T

←(SETQ A1 (ARRAY 5))
(1)
(2)
```

```
(3)
(4)
(5)
{ARRAYP}#542674
```

The effect of DEFPRINT is immediate once the function is defined to INTERLISP.

Note that each of the elements is NIL so only the index appears. Now, let's populate the array using the following CLISP construct.

```
←(FOR I FROM 1 TO 5 DO (SETA A1 I (ITIMES I 100)))
NIL

←(PRINT A1)
(1 . 100)
(2 . 200)
(3 . 300)
(4 . 400)
(5 . 500)
{ARRAYP}#542674
```

DEFPRINT combined with the PRINTOUT package (see Section 15.8) provides you with considerable flexibility in formatting output of extended LISP objects and structures.

15.1.6 Printing Unusual Data Structures

Some data structures cannot be printed and read in easily. These include reentrant and circular data structures. INTERLISP provides the HPRINT package to print these data structures. HPRINT is intended for use with the File Package for dumping the values of variables to files so that they may later be reloaded.

HPRINT will print and read back any structure containing user datatypes, arrays, hash tables, and unusual list structures. It does so by following all of the pointers to the lowest level of the structure and writing each item encountered during its passage. Data so written are surrounded by special read macro characters (see Section 14.4.3) to identify the unusual items in the structure when it is read back by INTERLISP.

The HPRINT package provides three functions for printing, reading, and copying unusual data structures. These take the following formats

```
Function:       HPRINT

# Arguments:    4

Arguments:      1) an S-expression, EXPRESSION
                2) a file name, FILE
```

```
                3) a circular list flag, UNCIRCULAR
                4) the type of data, DATATYPESEEN

Value:          NIL
```

HPRINT prints the value of EXPRESSION on FILE. If UNCIRCULAR has the value non-NIL, HPRINT will not check for circularities in EXPRESSION. If you do not check for circularities in your data structures, you may miss chances for greater speed and more efficient use of storage.

HPRINT is usually used to print structures to disk files. The algorithm relies on an ability to reset the file pointer to indicate duplicate elements in EXPRESSION. If FILE is not a disk file and UNCIRCULAR is NIL, HPRINT opens a temporary file, HPRINT.SCRATCH, prints the expression on it, and then copies that file to the primary output file. The temporary file is deleted.

Consider the following example:

```
←(SETQ rare-gases
(LIST 'helium 'krypton 'argon 'xenon 'radon))
(helium krypton argon xenon radon)
```

Let us make this list circular via the following statement:

```
←(RPLACD (LAST rare-gases) rare-gases)
(helium krypton argon xenon radon helium krypton argon
...)
```

which would continue to print forever.

Now, we can print this circular list via

```
←(HPRINT rare-gases T NIL)
(helium krypton argon xenon radon . [1])
NIL
```

where the CDR of the last cell of the list indicates that the foregoing elements repeat indefinitely.

When HPRINT sees a user datatype for the first time, it will print a summary of that datatype's declaration. When this is read in the datatype is redeclared. If DATATYPESEEN is non-NIL, HPRINT assumes that the same datatype declaration is to be used at read time that was used at print time, and does not print the declarations. Consider the example based on complex numbers (see Section 13.7.2):

```
←(SETQ x (COMPLEX 1.0 3.0))
((1.0 . 3.0))      because we already have a DEFPRINT
                   declaration in force
```

```
←(HPRINT x T NIL NIL)
{$COMPLEX (FLOATP FLOATP) 1.0 3.0]
←(HPRINT x T NIL T)
{COMPLEX 1.0 3.0)
```

To invoke HPRINT from the File Package, you may use the file package commands HORRIBLEVARS and UGLYVARS (see Section 17.2.2).

Reading Unusual Data Structures

The corresponding function for reading an unusual data structure is **HREAD**. It takes the form

```
Function:       HREAD
# Arguments:    1
Arguments:      1) a file name, FILE
Value:          NIL
```

HREAD reads in an unusual data structure from a file. Consider the following example:

```
←(OPENFILE 'JUNK 'OUTPUT)
⟨KAISLER⟩JUNK..1
←(HPRINT x 'JUNK NIL NIL)
NIL
←(CLOSEF 'JUNK)
⟨KAISLER⟩JUNK..1
```

Now, we can read the contents of the file using READFILE to see what has been printed out:

```
←(READFILE 'JUNK)
([$COMPLEX (FLOATP FLOATP) 1.0 3.0])
```

And, to set Y to the value in the file JUNK, we can use HREAD as follows:

```
←(OPENFILE 'JUNK 'INPUT 'OLD)
⟨KAISLER⟩JUNK..1
←(SETQ y (HREAD 'JUNK))
((1.0 . 3.0))
←(REAL y)
1.0
```

```
←(IMAG y)
3.0
```

Copying Unusual Data Structures

Finally, you may use the function HCOPYALL to copy an unusual data structure. It takes the form

```
Function:       HCOPYALL
# Arguments:    1
Arguments:      1) an S-expression, EXPRESSION
Value:          The value of EXPRESSION.
```

To set Z to the value of Y, we can use the expression

```
←(SETQ z (HCOPYALL y))
((1.0 . 3.0))
```

15.1.7 Writing Expressions to a File

You may write a sequence of one or more expressions to a file using **WRITEFILE**, which takes the form

```
Function:       WRITEFILE
# Arguments:    2
Arguments:      1) a list of one or more S-expressions,
                   EXPRESSION
                2) a file name, FILE
Value:          The file name.
```

WRITEFILE writes a date expression to FILE. It then writes the successive S-expressions that are the value of EXPRESSION to the file using FILERDTBL as its read table. If EXPRESSION is an atom, its value is used (for example, the name of a variable whose value is a list of functions). If FILE is not open, it is opened prior to writing.

If FILE is a list, (CAR FILE) is used as the name of the file to be written and it is left open. Otherwise, the atom STOP is written to the file after the value of EXPRESSION and the file is closed.

Consider the following example:

```
←(OPENFILE 'TEST 'OUTPUT)
⟨KAISLER⟩TEST..1
```

```
←(WRITEFILE COMPLEXCOMS 'TEST)
⟨KAISLER⟩TEST..1
```

Now, to check what has been written to the file, we may use READFILE (see Section 14.2.9).

```
←(READFILE 'TEST)
((PRIN1
     (QUOTE
          "WRITEFILE of ⟨KAISLER⟩TEST..1 MADE BY KAISLER
          ON 29-Jul-84 13:27:30" T)
 (FNS * COMPLEXFNS)
 (RECORDS COMPLEX)
 (P
     (DEFPRINT (QUOTE COMPLEX) (FUNCTION
     PRINT.COMPLEX))))
```

Note that READFILE does not return the atom STOP which signals the end of the file.

Terminating a File

ENDFILE merely writes STOP on the file and closes it. It takes the form

```
Function:       ENDFILE
# Arguments:    1
Argument:       1) a file name, FILE
Value:          The file name.
```

If FILE is not opened for output, ENDFILE generates an error.

15.2 PRINT CONTROL FUNCTIONS

INTERLISP provides several functions to control the positioning of information in an output line. These include printing multiple spaces, tabbing, and inserting carriage returns.

15.2.1 Printing Multiple Spaces

SPACES prints N spaces from the current position counter in the output line. It takes the form

```
Function:       SPACES
# Arguments:    2
```

```
Arguments:      1) the number of spaces, NUMBER
                2) a file name (optional), FILE

Value:          NIL
```

SPACES is subject to line length checking. The value returned by SPACES is NIL, so SPACES may be the last function called in a function. Assuming we begin in position 1:

```
←(PRIN1 ''Hello,")
←(SPACES 10)
←(PRIN1 ''Tom!")

12345678901234567890
Hello,          Tom!
```

A Definition for SPACES

A simple definition of SPACES (assuming printing is directed to the primary output file) might appear as

```
(DEFINEQ
     (spaces (number)
          (COND
               ((NOT (NUMBERP number))
                    (*
                         Generate an error if the
                         argument is not a number.
                    )
                    (ERROR "Argument not a number")
                    (BREAK))
               ((ZEROP number)
                    (*
                         Do nothing if zero spaces are
                         requested.
                    )
                    NIL)
               ((GREATERP number 0)
                    (*
                         Use a simple loop to PRIN1 the
                         blank character.
                    )
                    (PROG (a-number)
                          (SETQ a-number 0)
                    LOOP
                          (PRIN1 " ")
                          (AND
```

```
                    (IGREATERP a-number number)
                    (RETURN))
                (SETQ a-number (ADD1 a-number))
                (GO LOOP))))
    ))
```

5.2.2 Printing a Carriage Return

PRIN1 and PRIN2 both print their results without forcing a new line. The user must explicitly place a carriage return in the output buffer to start a new output line. **TERPRI**, for terminate printing, places a carriage return followed by a line feed in the output buffer. It takes the form

```
Function:       TERPRI
# Arguments:    1
Argument:       1) a file name (optional), FILE
Value:          NIL
```

Consider the following example:

```
←(PRIN1 "An example of TERPRI")
←(TERPRI)
```

would result in output appearing as

```
---------- beginning of the line
|
v
An example of TERPRI⟨CR⟩
⟨beginning of next output line⟩
```

TERPRI may take an optional file name whence the carriage return and line feed are placed in the file.

15.2.3 Tabbing

Most formatted output requires that the program be able to specify position of the output line in which to place the resulting characters. **TAB** allows you to move the position cursor to the proper place in the output buffer. Its format is

```
Function:       TAB
# Arguments:    1-3
```

```
Arguments:      1) a position, POS
                2) minimum spaces to print, MINSPACES
                3) a file name (optional), FILE
Value:          NIL
```

TAB moves the position cursor in the current output line to POS. MINSPACES indicates the minimum number of spaces by which the cursor is to be displaced. If MINSPACES is NIL, 1 is assumed.

If the current position plus MINSPACES is greater than POS, TAB executes TERPRI and then (SPACES POS). If MINSPACES is T and the current position is greater than POS, TAB does nothing.

Consider the following example:

```
←(PROGN (PRIN1 "HELLO,") (TAB 25) (PRIN1 "JERRY!"))
         11111111122222222223
123456789012345678901234567890
HELLO,                    JERRY!
```

But, consider using MINSPACES:

```
←(PROGN (PRIN1 "HELLO,") (TAB 25 20) (PRIN1 "JERRY!"))
         1111111112222222222
12345678901234567890123456789
HELLO,
                          JERRY!
```

because 20 plus the current position (6) yields 26, which is greater than the position specified for TAB. Thus, it prints a ⟨CR⟩⟨LF⟩ and spaces to the proper position on the following line.

A Definition for TAB

We might define TAB as follows:

```
(DEFINEQ
    (tab (pos minspaces file)
         (PROG (posit)
               (*
                   Determine character position in the
                   current line of the file.
               )
               (SETQ posit (POSITION file))
               (SPACES
                   (COND
```

```
                    ((IGREATERP
                         (IPLUS posit (OR minspaces
                         1))
                         pos)
                         (*
                              If the current
                              position plus
                              MINSPACES is greater
                              than the position to
                              tab to, move to the
                              next output line.
                         )
                         (TERPRI file)
                         pos)
                    (T
                         (*
                              Otherwise, just move
                              the requisite number
                              of positions.
                         )
                         (IDIFFERENCE pos x)))
               file))
))
```

15.3 SETTING THE PRINT LEVEL

Printing functions are affected by an internal parameter that determines the level to which lists are to be displayed on the terminal. This parameter is set by **PRINTLEVEL**, whose format is

Function:	PRINTLEVEL
# Arguments:	2
Arguments:	1) CAR print level value, CARLVL 2) CDR print level value, CDRLVL
Value:	The previous parameter values for print level.

The print level parameter determines how much of a list is displayed at the terminal when one of the printing functions is executed. The CAR print level controls the number of unpaired left parentheses that will be printed. Below that level, all lists are denoted by the symbol &. The CDR print level controls the number of list elements that will be printed. Remaining elements are indicated by -- followed by a right parenthesis. In effect, list printing is truncated. Initial values for the CAR and CDR print levels are 1000 and −1, respectively.

Consider the following examples:

```
←(SETQ SYSPRETTYFLG 'T)
NIL
←(SETQ states
       '(maryland
           (ohio idaho
                 (virginia
                     (new-york texas)
                  maine)
            oregon)
         iowa))
(maryland (ohio idaho (virginia (new-york texas) maine)
oregon) iowa)
```

With the inital setting, where 1000 implies a (virtually) infinite print level, SHOWPRINT displays the following output:

```
←(SHOWPRINT states)
(maryland (ohio idaho (virginia (new-york texas) maine)
oregon) iowa)
```

Now, let us set the print level for CAR to 2:

```
←(PRINTLEVEL (CONS 2 -1))
(1000 . -1)
←(SHOWPRINT states)
(maryland (ohio idaho & oregon) iowa)
←(PRINTLEVEL (CONS 2 2))
(2 . -1)
←(SHOWPRINT states)
(maryland (ohio --) --)
←(PRINTLEVEL (CONS 0 0))
&
```

because the print level also affects system output.

```
←(SHOWPRINT states)
&
```

You may change the CAR and CDR print levels independently by executing

```
(PRINTLEVEL n NIL)          change CAR print level
(PRINTLEVEL NIL n)          change CDR print level
```

If the CAR print level is negative, a carriage return will be inserted between all occurrences of a right parenthesis immediately followed by a left parenthesis.

```
←(SETQ fruits '(orange (apple lime) (lemon grape)))
(orange (apple lime) (lemon grape))

←(PRINTLEVEL -2 NIL)
(1000 . -1)

←(PRINT fruits)
(orange (apple lime)
(lemon grape))
```

If the CDR print level is negative, then actions concerning CDR printing are disabled, i.e., lists will be printed in their entirety.

PRINTLEVEL normally affects only terminal output. Usually, output to other files acts as though an infinite print level existed. You may force print levels to be applied to file output other than the terminal by setting PLVLFILEFLG to T. Initially, PLVLFILEFLG has the value NIL.

Dynamically Setting the Print Level

If some of your functions are particularly long, you may wish to shorten the printing cycle by redefining the print level. To do so while a function is printing, you must type CTRL-P followed by a number which is terminated by a period or an exclamation point. The CAR print level is immediately set to this number.

When a CTRL-P is typed, it causes an interrupt to INTERLISP. INTERLISP immediately takes the following steps:

1. Saves and clears the input buffer
2. Clears the output buffer
3. Rings the bell (see PRINTBELLS) to notify you that it has seen the CTRL-P
4. Waits until you enter a number for the new print level
5. Restores the input buffer
6. Continues printing

Upon return from the CTRL-P, the print level may have been changed. If the expression being printed is currently at a deeper level than the new print level, all unfinished lists are terminated by "--)". You may terminate a particularly long expression by typing CTRL-P 0.

If the number following the CTRL-P is followed by a comma, you may then type another number which specifies the new value of the CDR print level.

The period terminating the CTRL-P input indicates that the print level should be returned to its previous setting after completing the printing of the current expression. Terminating the CTRL-P input with an exclamation point (!) causes the change to be permanent (at least, until it is changed again!).

15.4 PRINTING NUMBERS

Numbers, like other atoms, have standard print names. The display of these names is sensitive to the values of two variables:

RADIX	which determines the basis of conversion for the representation of numbers.
FLTFMT	which specifies the floating point format.

PRINTNUM allows you to control the formatted printing of both fixed and floating point numbers. PRINTNUM takes the following form:

Function:	PRINTNUM
# Arguments:	3
Arguments:	1) a format specification, FORMAT 2) a number, NUMBER 3) a file, FILE
Value:	The print name of the number.

Format specifications are described in the following sections.

If NUMBER is not a number (e.g., not NUMBERP) and non-NIL, PRINTNUM displays the error message NON-NUMERIC ARG. If NUMBER is NIL, PRINTNUM uses the value of NILNUMPRINTFLG to decide what to do. If NILNUMPRINTFLG is NIL, an error is generated. Otherwise, the value of NILNUMPRINTFLG is displayed right-justified in the field.

If the full print name of NUMBER will not fit into the specified field, it will be printed in its entirety. Then, a TAB is executed to properly place the line position in the output file.

15.4.1 Format Conversion

PRINTNUM may use intrinsic operating system routines to assist in printing formatted numbers when the format can be specified by some sort of special code (e.g., INTERLISP-10 and INTERLISP/370). FORMAT is converted from its machine-independent list form into a sequence of codes that are used by operating system routines to print numbers of the appropriate type.

In most cases, your program will use a several different formats, but will use them repeatedly throughout its execution. Each time your program calls PRINTNUM, FORMAT will be converted anew. You may improve the efficiency of your program by converting the formats once and storing the resulting format code for usage by PRINTNUM. **NUMFORMATCODE** performs this conversion for you. It takes the form

Function:	NUMFORMATCODE
# Arguments:	2
Argument:	1) a format specification, FORMAT 2) a smashflag, SMASHFLAG
Value:	A format code for the particular operating system under which your INTERLISP is running.

Consider the following example (from INTERLISP-10):

```
←(NUMFORMATCODE '(FIX 3 NIL T))
(FIX . 15033171978)

←(NUMFORMATCODE '(FLOAT 11 2 2))
(FLOAT . 721436804)

←(NUMFORMATCODE)
(NIL . 10000)
```

Executing NUMFORMATCODE with no arguments returns an uninitialized data element that can later be smashed into by succeeding calls.

If SMASHFLAG is a format-code data structure, the new format code will be placed into that represented by SMASHFLAG rather than being allocated new storage.

Note that in INTERLISP-D this function is a no-op since there is no special internal representation for number formats.

15.4.2 Fixed Point Format

The FORMAT for a fixed point number is a list structure comprising the following items:

FIX	An identifier specifying format type.
⟨width⟩	The number of characters comprising the field in which to display the number.
⟨radix⟩	The basis for numeric conversion.
⟨pad⟩	If NIL, the field is padded with spaces. If T, the field is padded with "0" (zero).
⟨leftflush⟩	If NIL, the number is right-justified in the field. If T, the number is left-justified in the field with trailing spaces filling the field.

Numbers, if not FIXP, are rounded to the nearest integer. Consider the following examples:

Function Call	Result
(PRINTNUM '(FIX 2) 27)	27
(PRINTNUM '(FIX 3 NIL T) 6)	006
(PRINTNUM '(FIX 10 8 T) 10	0000000012
(PRINTNUM '(FIX 6 NIL NIL T) 23)	23bbb

where 'b' indicates a blank or space.

15.4.3 Floating Point Format

A floating point number is always printed as a decimal number. FORMAT is a list structure comprising the following items:

FLOAT	An identifier specifying format type.
⟨width⟩	The number of characters comprising the field in which to display the number; it must account for the decimal point.
⟨decpart⟩	The number of digits to the right of the decimal point. If NIL, ⟨decpart⟩ is assumed to be zero (e.g., no digits to the right of the decimal point).
⟨exppart⟩	If non-NIL, the number is printed using exponent notation. It specifies the field size of the exponent including E and a possible sign prefacing the exponent.
⟨pad⟩	If NIL, padding on the left of the field is with spaces; otherwise, '0' (zero) is used.
⟨round⟩	If non-NIL, it indicates the digit position at which rounding will take place.

Consider the following examples:

Function Call	Result
(PRINTNUM '(FLOAT 7 2) 15.375)	bb15.38

```
(PRINTNUM '(FLOAT 7 2 NIL T) 15.375)          0015.37
(PRINTNUM '(FLOAT 7 2 NIL NIL 1) 18.76)       20.00
(PRINTNUM '(FLOAT 11 2 2) 175.34)             bbbbb1.75E2
```

15.4.4 Changing the Integer Radix

The INTERLISP printing functions assume a radix (or base) of 10 for printing integers. You may change the radix to another base via **RADIX**. It takes the form

```
Function:       RADIX
# Arguments:    1
Argument:       1) an integer, N
Value:          The old radix.
```

RADIX resets the output radix for integers to the absolute value of N. If N is negative, integers are interpreted by the printing routines as unsigned numbers (e.g., as positive numbers in an infinite precision machine). Numeric output under a negative radix varies with the implementation. Consider the following examples (from INTERLISP-10):

```
←(RADIX)
10
←(PRINT -128)
-128
←(RADIX 8)
12Q
←(PRINT -128)
-200Q
←(RADIX 2)
1000
←(PRINT -128)
-10000000
←(RADIX -16)
2
←(PRINT -128)
???????80
```

because numeric output under a negative radix varies with implementation, and may be questionable in interpretation.

15.4.5 Changing the Floating Point Output Format

You may change the format for printing floating point numbers via **FLTFMT**, which takes the form

```
Function:       FLTFMT
# Arguments:    1
Argument:       1) a format specification, FORMAT
Value:          The current format.
```

FLTFMT sets the value of the variable FLTFMT to FORMAT which is a specification as described in Section 15.4.3. If FORMAT has the value T, the default floating point format is used. To set the value of FLTFMT. (FLTFMT) returns the current specification without changing it.

```
←(FLTFMT)
536870912
```

15.5 TERMINAL TABLES

A *terminal table* is used to specify the syntax classes associated with characters for output operations. Terminal tables may be made device dependent to account for the particular display features of different terminals. The system terminal table has the following values:

```
CHARDELETE      1               character deletion
LINEDELETE      17              line deletion code
RETYPE          18              retype line code
CTRLV           2
EOL             13              end-of-line code
RAISE           NIL             do not "raise" input
LINEDELETESTR   "##"            response to line delete
                                code

1STCHDEL        " "
NTHCHDEL        " "
POSTCHDEL       " "
EMPTYCHDEL      " "
ECHODELS?       NIL
CONTROL         NIL
0               REAL
```

```
1               IGNORE
2               IGNORE
3               IGNORE
4               IGNORE
5               IGNORE
6               IGNORE
7               SIMULATE
8               INDICATE
9               SIMULATE
10              SIMULATE
11              INDICATE
12              INDICATE
13              REAL
14              IGNORE
15              INDICATE
16              INDICATE
17              IGNORE
18              IGNORE
19              INDICATE
20              INDICATE
21              INDICATE
22              INDICATE
23              IGNORE
24              IGNORE
25              INDICATE
26              IGNORE
27              SIMULATE
28              INDICATE
29              INDICATE
30              INDICATE
31              SIMULATE
```

The functions GETSYNTAX, SETSYNTAX, and SYNTAXP (see Section 14.4.2) all operate with terminal tables. In general, the syntax class will define which type of table to use. When GETSYNTAX or SETSYNTAX are given tables with incompatible syntax classes, they generate error messages.

```
←(GETSYNTAX 'BREAK (GETTERMTABLE))
ILLEGAL READTABLE
[TERMTABLEP]#1,104740
←(SETSYNTAX 52 'CHARDELETE (GETREADTABLE))
ILLEGAL TERMINAL TABLE
[READTABLEP]#112203
```

A terminal table also contains information about terminal control including line buffering, character echoing, and case conversion.

Note that, unlike read tables, terminal tables cannot be passed to input or output functions. Consider what happens:

```
←(READ 'T (GETTERMTABLE))
ILLEGAL READTABLE
[TERMTABLEP]#1,104740
```

15.5.1 Terminal Syntax Classes

There are seven terminal syntax classes:

Class	Function	Association
CHARDELETE	Character Deletion	CTRL-A
LINEDELETE	Line Deletion	CTRL-Q
RETYPE	Reprint Line	CTRL-R
CTRLV		CTRL-V
EOL	End of Line	CR/LF
NONE	All other characters	none

Characters are assigned to a syntax class by SETSYNTAX (see Section 14.4.3). Assigning a character to a syntax class disables the previous character. That character is automatically assigned to NONE. That is, only one character may be assigned to the first five syntax classes mentioned above.

CHARDELETE deletes the previous character in the input buffer when it is typed. Repeated uses will delete successive characters back to the beginning of the buffer.

LINEDELETE deletes the current line when it is typed. It cannot be used repeatedly.

WORDDELETE deletes the previous "word" (e.g., symbol in the input buffer) where "word" is interpreted to be a sequnce of non-separator characters.

RETYPE causes the current line to be retyped when it is typed in. It is particularly useful when you have made repeated character and word deletions in a line.

CTRL-V causes the corresponding control character to be input when it is followed by the appropriate control character letter.

EOL signals to the line buffering routine that the line has been terminated when it is typed in. If there is an outstanding READ or READLINE, the contents of the buffer are transferred to the program.

```
1               IGNORE
2               IGNORE
3               IGNORE
4               IGNORE
5               IGNORE
6               IGNORE
7               SIMULATE
8               INDICATE
9               SIMULATE
10              SIMULATE
11              INDICATE
12              INDICATE
13              REAL
14              IGNORE
15              INDICATE
16              INDICATE
17              IGNORE
18              IGNORE
19              INDICATE
20              INDICATE
21              INDICATE
22              INDICATE
23              IGNORE
24              IGNORE
25              INDICATE
26              IGNORE
27              SIMULATE
28              INDICATE
29              INDICATE
30              INDICATE
31              SIMULATE
```

The functions GETSYNTAX, SETSYNTAX, and SYNTAXP (see Section 14.4.2) all operate with terminal tables. In general, the syntax class will define which type of table to use. When GETSYNTAX or SETSYNTAX are given tables with incompatible syntax classes, they generate error messages.

```
←(GETSYNTAX 'BREAK (GETTERMTABLE))
ILLEGAL READTABLE
[TERMTABLEP]#1,104740
←(SETSYNTAX 52 'CHARDELETE (GETREADTABLE))
ILLEGAL TERMINAL TABLE
[READTABLEP]#112203
```

A terminal table also contains information about terminal control including line buffering, character echoing, and case conversion.

Note that, unlike read tables, terminal tables cannot be passed to input or output functions. Consider what happens:

```
←(READ 'T (GETTERMTABLE))
ILLEGAL READTABLE
[TERMTABLEP]#1,104740
```

15.5.1 Terminal Syntax Classes

There are seven terminal syntax classes:

Class	Function	Association
CHARDELETE	Character Deletion	CTRL-A
LINEDELETE	Line Deletion	CTRL-Q
RETYPE	Reprint Line	CTRL-R
CTRLV		CTRL-V
EOL	End of Line	CR/LF
NONE	All other characters	none

Characters are assigned to a syntax class by SETSYNTAX (see Section 14.4.3). Assigning a character to a syntax class disables the previous character. That character is automatically assigned to NONE. That is, only one character may be assigned to the first five syntax classes mentioned above.

CHARDELETE deletes the previous character in the input buffer when it is typed. Repeated uses will delete successive characters back to the beginning of the buffer.

LINEDELETE deletes the current line when it is typed. It cannot be used repeatedly.

WORDDELETE deletes the previous "word" (e.g., symbol in the input buffer) where "word" is interpreted to be a sequnce of non-separator characters.

RETYPE causes the current line to be retyped when it is typed in. It is particularly useful when you have made repeated character and word deletions in a line.

CTRL-V causes the corresponding control character to be input when it is followed by the appropriate control character letter.

EOL signals to the line buffering routine that the line has been terminated when it is typed in. If there is an outstanding READ or READLINE, the contents of the buffer are transferred to the program.

15.5.2 Establishing a Terminal Table

You may establish a terminal table by executing **SETTERMTABLE**, which takes the form

```
Function:       SETTERMTABLE
# Arguments:    1
Argument:       1) the address of a terminal table, TTBL
Value:          The address of the previous terminal
                table.
```

If TTBL is not a real terminal table, SETTERMTABLE displays the error message "ILLEGAL TERMINAL TABLE". Otherwise, it sets the primary terminal table to TTBL. It returns the address of the previous terminal table.

15.5.3 Getting a Terminal Table Address

GETTERMTABLE obtains the address of a terminal table. It takes the form

```
Function:       GETTERMTABLE
# Arguments:    1
Argument:       1) the address of a terminal table, TTBL
Value:          TTBL, if it is a real terminal table;
                otherwise, an error message.
```

If TTBL is non-NIL and represents a real terminal table, the value of GETTERMTABLE is TTBL. If TTBL is not the address of a real terminal table, GETTERMTABLE displays the error message "ILLEGAL TERMINAL TABLE". In this case, GETTERMTABLE serves as a predicate to determine if TTBL is really a terminal table.

```
←(GETTERMTABLE)
[TERMTABLEP]#112406
```

When TTBL is NIL, GETTERMTABLE returns the address of the primary terminal table. This is the mode in which it is most often used. Generally, you will want to check if the current primary terminal table is the same as the original system terminal table or one you have created.

We might define the following function to check if the current terminal table is equal to the original system terminal table (e.g., the one supplied with the SYSOUT):

```
(DEFINEQ
     (origtermtable? (ttbl)
          (EQUAL origttbl
                 (COND
                    ((NULL ttbl)
                           (GETTERMTABLE))
                    (T ttbl)))
     ))
```

where ORIGTTBL was set upon entry to INTERLISP at session initiation via

```
(SETQ origttbl (GETTERMTABLE))
```

15.5.4 Testing a Terminal Table

You may test whether or not a data structure is a terminal table by executing **TERMTABLEP**, which takes the form

Function:	TERMTABLEP
# Arguments:	1
Argument:	1) the address of a terminal table, TTBL
Value:	TTBL, if it is a terminal table; otherwise, NIL.

It returns the address if the data structure is a terminal table, otherwise NIL. This function may seem redundant given the effect of GETTERMTABLE when it is supplied with a non-NIL argument. Note that TERMTABLEP returns NIL if its argument is not a valid terminal table whereas GETTERMTABLE causes an error.

```
←(TERMTABLEP (GETREADTABLE))
NIL
←(TERMTABLEP)
NIL
←(TERMTABLEP (GETTERMTABLE))
[TERMTABLEP]#112406
```

15.5.5 Copying Terminal Tables

You may copy a terminal table in order to modify it for your own use by executing **COPYTERMTABLE**, which takes the form

```
Function:       COPYTERMTABLE
# Arguments:    1
Argument:       1) the address of a terminal table, TTBL
Value:          The address of a new terminal table.
```

It returns the address of a new data structure that is a copy of the specified terminal table.

If TTBL is NIL or ORIG, a copy of the original system terminal table will be returned.

```
←(SETQ copyttbl (COPYTERMTABLE))
[TERMTABLEP]#543173
```

COPYTERMTABLE is the *only* function that can create a new terminal table for you. Moreover, you may only modify the terminal table via the function SETSYNTAX.

15.5.6 Resetting the Terminal Table

You may reset the terminal table from another terminal table using **RESETTERMTABLE**, which takes the form

```
Function:       RESETTERMTABLE
# Arguments:    2
Arguments:      1) the address of a terminal table, TTBL
                2) another terminal table, FROM
Value:          TTBL, if it is a terminal table.
```

RESETTERMTABLE copies FROM into TTBL. FROM and TTBL may be NIL or real terminal tables. FROM may have the value ORIG indicating that the original system terminal table should be used.

15.6 TERMINAL CONTROL

INTERLISP provides you with several functions that allow you to control the display portion of your terminal. These functions control the echoing of characters that you have typed at the keyboard.

15.6.1 Echo Modes

When a character is typed at the keyboard, low-level routines in the operating system read the character (usually via interrupts) and transfer it to the IN-

TERLISP terminal handler. Normal characters will be echoed immediately. Control characters, which are signified by pressing the *CONTROL* or *CTRL* key and another key simultaneously, may or may not be echoed.

Normally, control characters will be echoed using a format specified by ECHOCONTROL (see below). You may disable echoing for a terminal table by executing **ECHOMODE**, which takes the form

```
Function:       ECHOMODE
# Arguments:    2
Arguments:      1) an echo flag, ECHOFLAG
                2) a terminal table address, TTBL
Value:          The previous echo mode for TTBL.
```

Echoing is enabled when ECHOFLAG is T; otherwise, it is disabled. Note that changes in the echo mode do not take effect until the terminal table is made the primary terminal table (unless it is the primary terminal table). Consider the following example:

```
←(PRINT\T\1 'HELP)
HELPHELP
←(ECHOMODE NIL)
T
←HELPHELP
```

even though I typed (PRIN1 'HELP) because the input is not echoed, since I turned it off for the primary terminal table.

Obtaining the Current Echo Mode

You may determine the current echoing mode of a terminal table by executing **GETECHOMODE**, which takes the form

```
Function:       GETECHOMODE
# Arguments:    1
Argument:       1) the address of a terminal table, TTBL
Value:          The current echo mode for the indicated
                terminal table.
```

Consider the following example:

```
←(GETECHOMODE)
T
```

where we assume the terminal table has not been changed from the original system terminal table.

When TTBL is NIL, the primary terminal table is assumed:

```
←(ECHOMODE NIL)
T
```

will disable echoing on the primary terminal table.

15.6.2 Echo Control

ECHOCONTROL allows you to determine how control characters will be displayed at your terminal when they are typed in. The format of ECHOCONTROL is

Function:	ECHOCONTROL
# Arguments:	3
Arguments:	1) a character, CHAR 2) a mode, MODE 3) a terminal table address, TTBL
Value:	The previous mode for the character.

CHAR may be either a character or a character code. MODE takes one of the following values:

Value	*Effect*
IGNORE	The character is never printed.
REAL	The character is printed as itself.
SIMULATE	Output is simulated.
INDICATE	The character is printed, but preceded by an "up arrow".
NIL	The mode is not changed.

TTBL is the address of a terminal table for which the mode control is to be changed. If TTBL is NIL, the current terminal table is assumed. Any mode change does not take effect until a terminal table is made the current terminal table via SETTERMTABLE. Of course, if TTBL is NIL, the effect is immediate because the current terminal table is assumed.

You may force CONTROL-B to echo with an ↑ as follows:

```
←(ECHOCONTROL 'B 'INDICATE)
IGNORE
```

```
←⟨CTRL-B⟩↑B
BREAK
...
```

Echoing information may be specified for control characters only. Any other character will generate an error with the message "ILLEGAL ARG". The effect of this function applies to all displays of the control character including the printing of the control character from within a program as well as echoing it on input.

Note that different terminals respond to control characters in interesting ways. Some of the control characters may be intercepted and intepreted by the terminal itself, particularly with the new "smart" terminals. Some judicious experimentation may be necessary to make control characters acceptable at your terminal.

General Echo Control

INTERLISP-D and INTERLISP/VAX allow you to specify echoing for any character, not just control characters. **ECHOCHAR** takes the form

```
Function:       ECHOCHAR

# Arguments:    3

Arguments:      1) a character code, CHARCODE
                2) an echo mode, MODE
                3) a terminal table address, TTBL

Value:          The old mode for the character code.
```

ECHOCHAR allows you to specify echoing for any character code. CHARCODE may be a list of character codes, whence ECHOCHAR is applied to each character code with the same values of MODE and TTBL.

15.6.3 Character and Line Deletion Control

Two control characters, namely CTRL-A and CTRL-Q, control the deletion of characters and lines on typein. Normally, the system will backspace over the character (for CHARDELETE) or signify by printing ## to indicate that the respective deletions have taken place. You may modify the response to these two control characters to suit your application needs. In particular, you may want to display certain messages depending on the context in which a user attempts to delete characters.

To modify deletion control for a terminal table, you must execute **DELETECONTROL**, which takes the form

```
Function:       DELETECONTROL

# Arguments:    3
```

```
Arguments:      1) a type of deletion, TYPE
                2) a string, MESSAGE
                3) a terminal table address, TTBL

Value:          The previous message as a string.
```

TYPE specifies the output protocol to be observed when a deletion control character is typed in. TYPE may take one of the following values:

Value	Effect
LINEDELETE	MESSAGE is printed when a LINEDELETE control character is typed in. The initial setting is "##⟨cr⟩".
1STCHDEL	MESSAGE is printed the first time a CHARDELETE control character is typed. The initial setting is "\".
NTHCHDEL	MESSAGE is printed on subsequent entries of the CHARDELETE control character without intervening characters. The initial setting is " " (e.g., the null string).
POSTCHDEL	MESSAGE is printed when a character other than a control character is entered following character or line deletion. The initial setting is "\".
EMPTYCHDEL	MESSAGE is printed when a CHARDELETE control character is typed and the line buffer is empty. The initial setting is "##⟨cr⟩".
ECHO	Any characters deleted by CHARDELETE are echoed at the terminal.
NOECHO	Any characters deleted by CHARDELETE are not echoed at the terminal.

The combined effect of 1STCHDEL and NTHCHDEL is to display a "\" (e.g., a backslash) when the first of a sequence of characters is deleted and then the empty string. However, since ECHO is usually enabled for character deletion, the sequence of characters deleted will appear in reverse order on your display.

```
←(SETQ keystone.state 'pennslyv\vyl\ylvania)
                          ↑
                          |
                  Here I typed CTRL-As to
                  delete three characters.
```

Note that the deleted characters are both echoed and surrounded by "\" to delineate them from the correct text.

For all but, ECHO and NOECHO, the value of DELETECONTROL is the previous value of the message to be printed. ECHO and NOECHO return the previous mode. MESSAGE must be less than five characters due to timing considerations in responding to user inputs.

The IRM suggests that you may backspace over characters rather than echoing them if you have a video display terminal. To do so, you may specify the following sequence of commands:

```
(ECHOCONTROL 8 'REAL)          Specifies that CTRL-H, which
                               is backspace, will be
                               displayed as is.

(DELETECONTROL 'NOECHO)        Eliminates echoing of
                               deleted characters.

(DELETECONTROL '1STCHDEL "↑H ↑H")

(DELETECONTROL 'NTHCHDEL "↑H ↑H")
```

Obtaining the Current Deletion Control Message

You may determine the current deletion control message by executing **GETDELETECONTROL**. It takes the form

```
Function:       GETDELETECONTROL

# Arguments:    2

Arguments:      1) a type of deletion, TYPE
                2) a terminal table address, TTBL

Value:          The current deletion control message for
                TYPE in TTBL.
```

Consider the following examples:

```
←(GETDELETECONTROL 'LINEDELETE)
"##⟨CR⟩
"

←(GETDELETECONTROL 'NTHCHDEL)
" "
```

```
←(GETDELETECONTROL 'POSTCHDEL)
"\"
```

Note that when TTBL is NIL, the current terminal table is assumed.

15.6.4 Converting to Upper Case on Typein

INTERLISP accepts S-expressions and commands in upper-case format only (comments and a few other words being exceptions). Most computer systems, however, support both upper- and lower-case data entry. Many times, you will find it confusing to keep switching between lower and upper case. To remedy this, you may direct INTERLISP to convert all characters entered on typein to upper case. To do so, you must execute **RAISE**. It takes the form

```
Function:       RAISE
# Arguments:    2
Arguments:      1) a flag, FLAG
                2) a terminal table address, TTBL
Value:          The previous setting of FLAG.
```

If FLAG is T, all characters are echoed as they are typed in, but lower-case characters are converted to upper case when they are passed to INTERLISP functions. Otherwise, all characters are passed through exactly as they are typed.

Determining the Current Raise Mode

You may determine the current raise mode for a terminal table by executing **GETRAISE**, which takes the form

```
Function:       GETRAISE
# Arguments:    1
Argument:       1) a terminal table address, TTBL
Value:          The current RAISE mode for TTBL.
```

Consider the following example:

```
←(GETRAISE)
NIL
```

15.6.5 Line Length Control

The length of the output line is usually determined by the operating system. Standard line length on terminals is 80 columns, but most operating systems

allow you to vary the line length from 1 to 132 columns. INTERLISP provides access to the functions that allow you to vary terminal line length.

Setting the Terminal Line Length

SETLINELENGTH allows you to set the terminal line length. It takes the form

```
Function:       SETLINELENGTH
# Arguments:    1
Argument:       1) a line length, N
Value:          The former setting of TTYLINELENGTH.
```

If N is NIL, INTERLISP sets to the variable TTYLINELENGTH to the value of the terminal line length returned by the operating system. If N is not NIL, INTERLISP sets TTYLINELENGTH to N and calls upon the operating system to set the terminal line length to this value.

```
←(SETLINELENGTH)
72
```

which determines the current logical record size of the terminal.

Now, if we set the terminal line length to 10,

```
←(SETLINELENGTH 10)
72
←TTYLINELENGTH
10
←(PRINT "The quick red fox jumped over the lazy brown dog")
"The quick
 red fox j
umped over
 the lazy
brown dog"
```

Notice that INTERLISP automatically inserted ⟨LF⟩⟨CR⟩ after the proper number of characters while printing the string.

You should note that TTYLINELENGTH also affects typing in input commands, but I have not shown this feature in order to make the example above more readable.

Setting the File Line Length

A corresponding function is **LINELENGTH**, which allows you to set the line length (i.e., the logical record length) for an output file. It takes the form

```
Function:       LINELENGTH

# Arguments:    2

Arguments:      1) a line length, N
                2) a file name, FILE

Value:          The former setting of FILELINELENGTH.
```

LINELENGTH sets the logical record size of the output file FILE to N. If FILE is NIL, INTERLISP assumes that you meant the primary output file. The file must be open for output in order to determine or set the line length.

```
←(LINELENGTH NIL 'COMPLEX)
FILE NOT OPEN
COMPLEX
```

Let us open the file first:

```
←(IOFILE 'COMPLEX)
⟨KAISLER⟩COMPLEX..9

←(LINELENGTH NIL 'COMPLEX)
72
```

Note that if N is NIL, LINELENGTH determines the current logical record size of the file.

When a file is first opened, its line length is set to the value of FILELINELENGTH. You may reset the line length at any time. You may also adjust the value of FILELINELENGTH for all newly opened files by resetting FILELINELENGTH via SETQ.

Determining the Read or Write Position

POSITION allows you to determine the character position in the current input or output line of a file. It takes the form

```
Function:       POSITION

# Arguments:    2

Arguments:      1) a file name, FILE
                2) a column number, N

Value:          The previous column number.
```

If N is NIL, POSITION determines the column number of the next character to be read or printed on the specified file. Note that the file must be opened for input or output. If FILE is NIL, INTERLISP assumes that you mean the current primary input or output file.

```
←(POSITION)
0
```

A 0 is returned after an EOL has been read or printed indicating the start of a new line. If N is non-NIL, the column number is reset to the value of N.

```
←(IOFILE 'TEST)
⟨KAISLER⟩TEST..1

←(POSITION 'TEST 20)
0

←(PRIN1 "BEGIN AT COLUMN 20")
"BEGIN AT COLUMN 20"

←(CLOSEF 'TEST)
⟨KAISLER⟩TEST..1
```

And TEST would have the appearance on disk

```
                    22222222233333333334
12345678901234567890123456789012345678890
             BEGIN AT COLUMN 20
```

15.7 PRETTYPRINTING

The syntax of S-expressions often becomes an obtuse jumble of symbols and parentheses. If you are not careful, running together pieces of different expressions makes your code unreadable. *Prettyprinting* is the process of displaying S-expressions in a structured format that makes them both readable and understandable.

INTERLISP includes three functions for handling prettyprinting chores.

15.7.1 Generalized Prettyprinting

PRETTYPRINT is the standard system prettyprinting function. It takes the form

```
Function:       PRETTYPRINT

# Arguments:    2

Arguments:      1) a list of functions, FNS
                2) a pretty definition flag, PRETTYDEF

Value:          The value of FNS.
```

PRETTYPRINT always directs its output to the current primary output file (which was set by OUTPUT if not the default). Consider the following example:

```
←(PRETTYPRINT '(depth))
(depth
     (LAMBDA (lst)
          (COND
               ((NULL lst) 0)
               ((ATOM (CAR lst))
                    (depth (CDR lst)))
               ((GREATERP (ADD1 (depth (CAR lst)))
                          (depth (CDR lst)))
                    (ADD1(depth (CAR lst))))
               (T
                    (depth (CDR lst)))))
```

You will note that we have prettyprinted the functions and examples described in this text because it makes them both readable and understandable as well as pleasing to the eye.

PRETTYPRINT works on functions that have been broken or advised. It also works on functions which have been compiled but whose source has been saved on their property list.

If PRETTYPRINT is given an atom which is not the name of a function, but which has a value, it will treat the value of the atom as a sequence of function names.

```
←(SETQ x '(EQUAL (CAR y) (CDR z)))
(EQUAL (CAR y) (CDR z))

←(PRETTYPRINT x)
(EQUAL not printable)
((CAR y) not printable)
((CDR z) not printable)
(EQUAL (CAR y) (CDR z))
```

Functions, whose names are given to PRETTYPRINT, may not have their definitions loaded into memory. However, if the function resides in a file that is currently noticed by the File Package, INTERLISP will load the function definition using LOADFNS (see Section 17.9.2). It will then print the definition as described above.

As a last resort, PRETTYPRINT attempts spelling correction on an argument. If this fails, PRETTYPRINT returns the error message (⟨atom⟩ NOT PRINTABLE).

PRETTYDEFFLG has the value T when PRETTYPRINT is called from PRETTYDEF (see Section 17.8.1). This flag causes PRETTYPRINT to print the name of the current function it is writing to the output file as an indication of its progress.

A Definition for PRETTYPRINT

We might define PRETTYPRINT as follows:

```
(DEFINEQ
    (prettyprint (lst prettydefflg)
        (PROG (expression fn definition)
            (COND
                ((ATOM lst)
                    (*
                        If given an atom, listify
                        it so that PRETTYPRINT may
                        attempt to work correctly.
                    )
                    (SETQ lst (LIST lst))))
            loop
            (COND
                ((NULL lst)
                    (RETURN NIL))
                ((AND prettydefflg
                    (NEQ (OUTPUT) T)
                    (*
                        If the current output file
                        is not the terminal, print
                        the first function name on
                        the terminal to let the
                        user know what progress we
                        are making in printing the
                        list of functions.
                    )
                    (PRINT (CAR lst) T)))
            (SETQ fn (CAR lst))
            loop1
            (*
                Get the definition of the function
                for printing.
            )
            (SETQ definition
                (COND
                    ((CDR fn)
                        (VIRGINFN fn))
                    (T
                        (GETD fn))))
             (COND
                ((AND (NULL definition)
                      (SETQ expression
```

```
                                (FNCHECK fn
                                         T
                                         prettydefflg)))
                            (RPLACA lst (SETQ fn
                            expression))
                            (GO loop1))
                       ((NULL (EXPRP definition))
                            (*
                                 If the function does not
                                 have an definition or an
                                 EXPR property, then it is
                                 not printable.
                            )
                            (PRIN1 fn)
                            (SPACES 1)
                            (PRIN1 "not printable")
                            (TERPRI)
                            (RETURN NIL))
                       (T
                          (*
                               FN is not the name of a
                               function, but it may be
                               misspelled, so attempt
                               spelling correction if DWIM
                               is enabled.
                          )
                          (AND DWIMFLG
                               (SETQ fn (ADDSPELL fn)))
                          (*
                               CLISPIFY the definition as
                               well.
                          )
                          (SETQ definition
                                (CLISPIFY definition)))
                          (TERPRI)
                          (PRIN1 '%())
                          (PRINT fn)
                          (PRINTDEF definition)
                          (PRIN1 '%)))
                          (TERPRI)))
                    (SETQ lst (CDR lst))
                    (GO loop))
))
```

15.7.2 Prettyprinting to the Terminal

PP prettyprints the value of its argument to the terminal. It takes the form

```
Function:       PP
# Arguments:    1-N
Arguments:      1) a function name, FN
                2-N) function names, FN[2]
                     ... FN[N]
Value:          A list of the function names.
```

PP is an NLAMBDA, nospread function. Consider the following example:

```
←(PP 'REAL 'IMAG)
(REAL
     (LAMBDA (CX) **COMMENT**
     (RECORDACCESS (QUOTE REAL)
                   CX NIL (QUOTE FETCH))))

(IMAG
     (LAMBDA (CX) **COMMENT**
     (RECORDACCESS (QUOTE IMAG)
                   CX NIL (QUOTE FETCH))))

(REAL IMAG)
```

In effect, PP could be defined as follows:

```
(DEFINEQ
    (pp (fn)
        (OUTPUT T)
        (SETREADTABLE T)
        (LINELENGTH TTYLINELENGTH)
        (PRETTYPRINT fn)
    ))
```

A much better definition of PP appears as follows:

```
(DEFINEQ
    (pp
        (NLAMBDA fnslst
            (PROG ((y (OUTPUT T)))
                  (OR
                      (ERSETQ
```

```
                                    (PRETTYPRINT
                                       (COND
                                          ((ATOM fnslst)
                                                (LIST fnslst))
                                          (T fnslst))))
                                 (TERPRI T))
                              (OUTPUT y)
                              (RETURN fnslst))
      )))
```

Note that this definition saves and restores the primary output file while printing the specified function to the terminal.

PP treats unknown functions in a fashion similar to PRETTYPRINT.

15.7.3 Displaying Prettyprinted Definitions from a File

PF allows us to display a prettyprinted definition from a file without actually loading the definition into memory. Note that both PRETTYPRINT and PP will load a function prior to displaying it if the function is noticed on a file. PF merely copies the bytes from a file to another file (typically the terminal without consuming space to store the function definition. It takes the following format:

```
Function:       PF

# Arguments:    3

Arguments:      1) a function name, FN
                2) a list of file name, FROMFILES
                3) a file name, TOFILE

Value:          NIL
```

PF is an NLAMBDA, nospread function. PF copies the definition of FN from each file named in FROMFILES to TOFILE. There are two cases:

1. If TOFILE is NIL, PF displays the definition(s) on the terminal.
2. If FROMFILES is NIL, PF invokes WHEREIS (see Section 17.3.8) to locate the file definition (if it resides in any of the noticed files).

Consider the following example:

```
←(PF COMPLEX COMPLEX T)
[from ⟨KAISLER⟩COMPLEX..9]
(COMPLEX
     (LAMBDA (R I) **COMMENT**
...
NIL
```

When PF prints to a terminal, it transforms the characters read from the file. Transformations occur because source definitions may be written to files in a variety of formats. PF does the following:

1. Removes font information.
2. Does not print the *change character* that marks revisions to the function definition.
3. Reduces the left margin to accommodate long expressions that may have been written to the file (e.g., those exceeding TTYLINELENGTH).
4. Suppresses comments, if **COMMENT**FLG is non-NIL.

15.7.4 Prettyprinting Symbolic Files

A generalized prettyprinting function is **PRINTDEF**, which takes the form

```
Function:       PRINTDEF

# Arguments:    6

Arguments:      1) an S-expression, EXPRESSION
                2) a left-hand margin, LEFT
                3) a definition flag, DEFFLAG
                4) a tail flag, TAILFLAG
                5) a list of functions, FNSLST
                6) a file name, FILE

Value:          The value of EXPRESSION.
```

PRINTDEF is described in more detail in Section 17.8 because it is used to place symbolic information in files.

15.7.5 Prettyprinting Control Variables

You may set the values of several variables to control the appearance of the prettyprinted output:

```
#RPARS          Determines how many right
                parentheses are necessary for
                substitution by square brackets.
                #RPARS is initialized to 4. If
                #RPARS is set to NIL, no square
                brackets will be substituted for
                right parentheses.
```

Consider the following example:

```
←(SETQ #RPARS NIL)
NIL
```

```
←(PP 'power)
(power
     (LAMBDA (x y)
          (COND
               ((ZEROP Y) 1)
               (T
                    (TIMES x
                           (power x (SUB1 y)))))
     ))
(POWER)
```

But, if we reset #RPARS

```
←(SETQ #RPARS 3)
3
←(PP 'power)
(power
     (LAMBDA (x y)
          ...
          (power x (SUB1 y)
(POWER)
```

#CAREFULCOLUMNS	PRETTYPRINT approximates the number of characters in each atom rather than actually counting them. This approach work well in most cases, but unusually long atom names will cause your output to assume a rather ragged appearance because PRETTYPRINT wraps around to the next line. Setting #CAREFULCOLUMNS to a non-zero, non-negative number causes PRETTYPRINT to compute the number of characters from the right hand margin. A value of 20 is probably sufficient. #CAREFULCOLUMNS is initialized to 0.
CHANGECHAR	When PRETTYPRINT prints to the terminal, you may annotate the listing to show changes made to

the S-expressions. PRETTYPRINT displays the value of CHANGECHAR, if non-NIL, in the right-hand margin for all S-expressions that the editor has marked as changed. CHANGECHAR initially has the value |.

CLISPIFYPRETTYFLG — If this variable is non-NIL, PRETTYPRINT invokes CLISPIFY to transform the S-expression before printing them (see Chapter 23).

```
←(SETQ CLISPIFYPRETTYFLG T)
T

←(PP POWER)
(power
     (LAMBDA (x y)
     (if Y=0
         then 1
         else (TIMES X (POWER x Y-1)
(POWER)
```

COMMENTFLG — PRETTYPRINT compares the CAR of each S-expression to this variable. If they are EQ, the S-expression is treated as a comment (see Section 16.10). COMMENTFLG initially has the value *.

FIRSTCOL — Specifies the starting column for printing comments. Its initial value is 48. (See Section 16.10 for a discussion of how comments are printed.)

PRETTYPRINTMACROS — Its value is an association list that allows you to format selected S-expressions. PRETTYPRINT compares the CAR of each S-expression to be printed with the CAR of the entries on the association list. If a match is found, the CDR of the association list entry is

	applied to the S-expression. If the result is NIL, PRETTYPRINT ignores the S-expression; otherwise, it prints it in the normal fashion. Thus, you may substitute one S-expression for another during printing. Its value is initially NIL.
PRETTYPRINTYPEMACROS	Its value is a list of elements having the form (⟨typename⟩ . ⟨function⟩) For each datatype other than lists and atoms, the type name is looked up on PRETTYPRINTYPEMACROS. If found, the function is applied to the expression. This allows you to format different datatypes such as arrays. Its value is initially NIL.

For example, to print an array using PP, we could define PRETTYPRINTYPEMACROS as follows:

```
←(SETQ PRETTYPRINTYPEMACROS '(ARRAYP . PRINT.ARRAYP))
(ARRAYP . PRINT.ARRAY)
```

PRETTYEQUIVLST	Its value is an association list to treat certain S-expressions like other S-expressions. For example, many File Package functions define forms using NLAMA. You can tell PRETTYPRINT to treat these expressions like NLAMBDAs using the following form: (NLAMA . NLAMBDA)
PRETTYFLG	If PRETTYFLG is NIL, PRINTDEF will use PRIN2 to print an expression instead of prettyprinting it. It is initially set to T.

PRETTYTABFLG	File space is often of significant concern to many users. The prettyprinting functions normally put spaces in the file to properly align the output, but at the cost of substantial quantities of space. You may save this space by replacing the spaces with tabs (assuming one tab is equivalent to 8 spaces). Tabs are not used if PRETTYTABFLG is NIL; it is initially set to T.

15.8 THE PRINTOUT PACKAGE

INTERLISP provides a set of standard printing functions including

PRIN1, PRIN2, PRINT	print an object on the specified output file
TAB	position the cursor at a specified location in the current print line
SPACES	insert 1 or more spaces at the current print position
PRINTNUM	print numbers in appropriate format
PRINTDEF	prettyprint expressions

By executing various combinations of these functions, you may format your output in any number of ways. To simplify the creation of certain formats, INTERLISP provides the Printout Package which implements a formatting language that specifies complicated sequences of the basic printing functions.

The Printout Package is implemented through CLISP (see Chapter 23). PRINTOUT is a CLISP word which is translated by DWIMIFY to the appropriate set of functions that achieve the desired effect. PRINTOUT takes a description of the desired format and prints it on the specified output file. A PRINTOUT description has the form:

```
(PRINTOUT ⟨file⟩
          ⟨printcom1⟩
          ⟨printcom2⟩
          ...
          ⟨printcomN⟩)
```

The ⟨file⟩ parameter specifies the file to which the output will be sent. The terminal must be explicitly specified as T in this form. The ⟨printcom⟩ parameters are forms that describe the individual printing events. A ⟨printcom⟩ may be a Printout Package command or an S-expression that is evaluated and whose value is to be printed.

If ⟨file⟩ is not the primary output file, the primary output file is "remembered" and ⟨file⟩ becomes the primary output for the duration of the PRINTOUT expression execution.

PRINTOUT commands may be divided into several logical categories. Following the IRM, these commands are discussed in the following sections.

15.8.1 Horizontal Spacing Commands

A *horizontal spacing command* provides a mechanism for moving the cursor to a new position within the current print line. In general, these commands will be translated to expressions that use TAB or SPACES. The horizontal spacing commands are

⟨n⟩	⟨n⟩ is a positive integer. The cursor is moved to the absolute position ⟨n⟩ within the current print line. If the print line cursor lies beyond ⟨n⟩, the file will be positioned at ⟨n⟩ on the next line (e.g., following an implicit TERPRI).
←n⟩	A negative integer indicates relative spacing forward in the print line. The print line cursor is moved forward ⟨n⟩ positions.
.TAB ⟨position⟩	This command is equivalent to ⟨n⟩, but makes the PRINTOUT expression easier to read and understand. ⟨position⟩ is an expression that is evaluated to produce an integer that is the absolution position. The print line cursor is moved to the specified position. If the print line cursor lies beyond the value of ⟨position⟩, the file will be placed at the value of ⟨position⟩ on the next line.
.TABO ⟨position⟩	Similar to .TAB but the value of ⟨position⟩ may be zero whence the print line cursor is not moved.

	One, two, or three apostrophes are a shorthand mechanism for specifying that one, two, or three spaces should be inserted in the current print line. They are equivalent to -1, -2, -3 respectively.
.SP ⟨distance⟩	Moves the print line cursor a total of ⟨distance⟩ spaces where ⟨distance⟩ is an S-expression to be evaluated. If ⟨distance⟩ is an integer, then .SP n and ←n⟩ are equivalent.
.RESET	Resets the current line by moving the print line cursor to the beginning of the line (via a carriage return without a line feed). You may use .RESET to position the cursor for overprinting of a line.

15.8.2 Vertical Spacing Commands

A *vertical spacing command* inserts a number of carriage returns or form-feeds in the output file. If you are printing to T, the effect is to see your output move up the screen. The vertical spacing commands are:

T	Inserts a carriage return and line feed into the file. It is equivalent to executing (TERPRI). The same effect may be achieved by inserting a 0 as a PRINTOUT command as this forces the file to move to position 0 (i.e., column 1) of the next line.
.SKIP ⟨lines⟩	Inserts a number of carriage returns into the output file. ⟨lines⟩ is an S-expression that is evaluated to produce the distance to be skipped.
.PAGE	Inserts a form-feed into the output file that causes a page to be ejected to the top of form when it is encountered.

15.8.3 Printing Specifications

PRINTOUT will use PRIN1 to print the values of expressions that are contained in a PRINTOUT form. PRIN1 may not be appropriate for every expression.

Additional printing commands are provided to allow you to choose alternative modes of output. The printing specifications are

.P2 ⟨expression⟩	The value of ⟨expression⟩ is printed using PRIN2.
.PPV ⟨expression⟩	The value of ⟨expression⟩ is prettyprinted beginning at the current print line position using PRINTDEF. PPV treats ⟨expression⟩ as a variable rather than a function.
.PPF ⟨expression⟩	The value of ⟨expression⟩ is prettyprinted as part of a function definition. Various INTERLISP functions that identify significant expressions in a function (such as PROG, SELECTQ, CLISP words) receive special formatting treatment.
.PPVTL ⟨expression⟩	The value of ⟨expression⟩ is printed as a tail, i.e., without the surrounding parentheses if it is a list. It is often used for prettyprinting sublists of a list whose other elements are formatted with other commands.
.PPFTL ⟨expression⟩	The value of ⟨expression⟩ is prettyprinted as a tail, but is treated as part of a function definition.

15.8.4 Structure Specifications

The prettyprinting routines will display the structure of S-expressions but are not designed to handle formatted text. Text is generally printed as a series of paragraphs whose left and right margins are defined to alter its shape. If you think of the page as a window in which you place blocks of text, then much of the time you will try to place your text so that is both appealing and pleasant to the reader as well as functional in purpose. The following commands are concerned with the shape and placement of a block of text.

Paragraph Specifications

You may specify a paragraph using the .PARA command. It takes the form

```
.PARA ⟨lmargin⟩ ⟨rmargin⟩ ⟨list⟩
```

where ⟨list⟩ is printed as a paragraph within the boundaries specified by ⟨lmargin⟩ and ⟨rmargin⟩, respectively. In general, ⟨list⟩ is a list of individual atoms arrayed as a stream to be printed. .PARA uses PRIN1 to print the list. An alternate form, .PARA2, uses PRIN2 instead.

Right Margin Flushing

In general, the PRINTOUT package will print paragraph lines with a right jagged edge. Many technical publications require contents of a paragraph to be flush against the right margin. Usually, text processors accomplish this by hyphenating words or introducing spaces between words to ensure that the line appears to be properly proportioned. The Printout Package allows you to perform right-flushing using the .FR command. It takes the form

```
.FR ⟨position⟩ ⟨expression⟩
```

where ⟨position⟩ determines the position where the right end of ⟨expression⟩ will line. ⟨position⟩ is interpreted as follows:

1. 0 indicates the right margin
2.)0 indicates the absolute position in the line
3. ⟨0 indicates a number of positions to the right of the current position

An alternative form, .FR2, uses PRIN2 rather than PRIN1 to print the value of the expression.

Centering an Expression

Titles and other significant expressions are usually centered in a line to draw attention to their content. You may center an expression using the .CENTER command. It takes the form

```
.CENTER ⟨position⟩ ⟨expression⟩
```

which centers the value of ⟨expression⟩ between the current position and ⟨position⟩. ⟨position⟩ is interpreted as described in 15.8.4.2.

An alternative form, .CENTER2, uses PRIN2 to print the value of ⟨expression⟩.

Printing Numbers

When you print numbers using the INTERLISP printing functions, they do their own internal formatting of the string representing the number. We described the Printnum Package in Section 15.4; it allows you to specify different formats for printing both integer and floating point numbers. The Printout Package allows

you to use some of the features of the Printnum Package via the following commands:

.I⟨format⟩ ⟨number⟩	Print ⟨number⟩ as an integer using PRINTNUM with a FIX format list constructed from ⟨format⟩.
.F⟨format⟩ ⟨number⟩	Print ⟨number⟩ as a floating point number using PRINTNUM with a FLOAT format list constructed from ⟨format⟩.

Note that the format specification immediately follows the command name.

```
.I5.-8.T
```

is translated to the format list (FIX 5 -8 T).

Individual elements of the format specification are separated from each other by periods.

.N ⟨format⟩ ⟨number⟩	Both .I and .F translate to PRINTNUM calls with quoted format specifications. .N allows you to call PRINTNUM with an expression which is evaluated to produce a format specification.

Note that the expression which produces the format is a separate list element rather than immediately following the .N command.

16

File Management and Operations

Most programs require external storage for data. To accomplish this, INTERLISP provides a set of functions for declaring external files to the system and performing operations upon them. This chapter discusses the basic file management functions and operations.

Most of the input/output functions described in the previous chapters may specify an optional file name as an argument. Most of these function require that the file already be open before it can be operated upon. INTERLISP maintains an internal data structure which describes open files and their status. Some implementations of INTERLISP utilize the capabilities of the underlying operating system to provide file management services. Thus, certain features of the file management software may not be available in every implementation.

16.1 FILE STRUCTURES AND NAMES

INTERLISP uses the file management functions of the host operating system whenever possible. In INTERLISP/370, the file management functions of the VM/SP operating system are used. Similarly, INTERLISP-10 and INTERLISP/VAX use the features of TENEX and UNIX, respectively. In INTERLISP-D, which provides its own operating system, many of these functions are encoded into the virtual machine that supports INTERLISP-D.

16.1.1 The VM/SP File System

The VM/SP file system is a flat file system. Files reside on minidisks which are logical entities that are subsets of a physical disk drive. VM/SP manages one or more minidisks for each user. Minidisks are identified by letters. The primary minidisk associated with a user identifier is labeled A. Subsequent disks are labeled B, C, D, and so on. Depending on the configuration data, a user may have from 1 to 26 minidisks accessible at any time, although he may have more minidisks associated with his virtual machine.

File names in the VM/SP operating system consist of up to eight characters, of which the first cannot be a number but all others may be; an eight-character file type such as FORTRAN, MODULE, etc.; and a minidisk identifier. Subdirectories and rooted tree structures common in other operating systems are not possible under VM/SP. You should consult the appropriate IBM manuals for further details about the VM/SP operating system.

16.1.2 The INTERLISP-D and INTERLISP-10 File Systems

INTERLISP-D supports a flexible file system where files may reside on the disk local to the Xerox 11xx Scientific Information Processor or on a file server accessible through the Ethernet.

Each file has a name that may consist of up to 40 alphanumeric characters including the punctuation marks +—$. Blanks are not allowed in a file name. Upper- and lower-case characters may be intermixed in a file name. However, they are not distinguished. INTERLISP-D translates all file name characters to upper case internally in order to operate upon them. When INTERLISP-D creates a file name, it will do so in upper-case letters.

Users of the TENEX version of INTERLISP will recognize that the file names have the same form except for a device/host name prefix. In INTERLISP-10, the file name may be prefixed by a directory name which is enclosed in angle brackets. Thus, a file name has the following format:

INTERLISP-D:

```
{device/host name}file name.file type;version
```

INTERLISP-10:

```
⟨directory-name⟩file name.file type;version
```

Files that reside on a Xerox 11xx processor's local disk belong to the device/host [DSK]. Files residing on other hosts will have the name of that host prefixed to the file name. For example, if files resided on a host whose name is RESEARCH, then to access those files you would prefix the host name [RESEARCH] to the file name.

File types are used to distinguish instances of a file name according to its contents, e.g., if you are working on a major INTERLISP project, you will probably keep several files containing various sorts of information. This information includes a SYSOUT which is the executable file, a LISPIN containing your program's source code, and perhaps a DATA file. By appending the file type (either SYSYOUT, LISPIN, or DATA) to the file name, you can distinguish the contents of the file and yet maintain uniformity in your file naming conventions.

The version of a file merely marks the current instance of the file. Any number of versions may reside on the disk or the file server. The INTERLISP func-

tions and predicates that manipulate files always retrieve the latest version of the file unless a version is explicitly specified in the argument to one of these functions. It is good practice to always keep at least one or two old versions of a file on the disk. More than that tends to clutter the disk and consume space that may otherwise be needed for data.

Unlike INTERLISP-10, [DSK] and the file servers do not support temporary files. Files that are created remain in place until you specifically delete them. However, you can simulate temporary files by defining them to reside in your virtual memory. To do so, prefix the file name with the host name [CORE]. All files defined in this manner reside within the user's virtual memory; they are saved with a SYSOUT of your virtual memory and are only lost when your virtual memory is abandoned.

16.2 FILE DECLARATION

Before a program can use a file, it must declare it to the system. A file may be declared to be input, output, or both upon its opening. After a file has been opened, the program may specify a file to be either its primary input or output file. When this occurs, all input will be accepted from or output directed to that file until a new declaration is executed.

16.2.1 The Primary File 'T'

When an INTERLISP program is initiated, it has associated with it a *primary file* for input and output operations. This file is commonly known as the file 'T' and is associated with the user's terminal. Until this primary file is changed, all input is accepted from and all output is directed to the user's terminal unless an optional file name is specified in the I/O function. For example, executing the PRINT function produces the indicated string on the user's terminal:

```
←(PRINT "Welcome to INTERLISP")
"Welcome to INTERLISP"
```

T is always considered to be open.

16.2.2 Declaring the Primary Input File

You may declare new primary files for input or output using two functions provided by INTERLISP.

To declare a new primary input file, you use **INPUT**, which takes the form

```
Function:       INPUT
                INFILE

# Arguments:    1
```

File names in the VM/SP operating system consist of up to eight characters, of which the first cannot be a number but all others may be; an eight-character file type such as FORTRAN, MODULE, etc.; and a minidisk identifier. Subdirectories and rooted tree structures common in other operating systems are not possible under VM/SP. You should consult the appropriate IBM manuals for further details about the VM/SP operating system.

16.1.2 The INTERLISP-D and INTERLISP-10 File Systems

INTERLISP-D supports a flexible file system where files may reside on the disk local to the Xerox 11xx Scientific Information Processor or on a file server accessible through the Ethernet.

Each file has a name that may consist of up to 40 alphanumeric characters including the punctuation marks +—$. Blanks are not allowed in a file name. Upper- and lower-case characters may be intermixed in a file name. However, they are not distinguished. INTERLISP-D translates all file name characters to upper case internally in order to operate upon them. When INTERLISP-D creates a file name, it will do so in upper-case letters.

Users of the TENEX version of INTERLISP will recognize that the file names have the same form except for a device/host name prefix. In INTERLISP-10, the file name may be prefixed by a directory name which is enclosed in angle brackets. Thus, a file name has the following format:

INTERLISP-D:

```
{device/host name}file name.file type;version
```

INTERLISP-10:

```
⟨directory-name⟩file name.file type;version
```

Files that reside on a Xerox 11xx processor's local disk belong to the device/host [DSK]. Files residing on other hosts will have the name of that host prefixed to the file name. For example, if files resided on a host whose name is RESEARCH, then to access those files you would prefix the host name [RESEARCH] to the file name.

File types are used to distinguish instances of a file name according to its contents, e.g., if you are working on a major INTERLISP project, you will probably keep several files containing various sorts of information. This information includes a SYSOUT which is the executable file, a LISPIN containing your program's source code, and perhaps a DATA file. By appending the file type (either SYSYOUT, LISPIN, or DATA) to the file name, you can distinguish the contents of the file and yet maintain uniformity in your file naming conventions.

The version of a file merely marks the current instance of the file. Any number of versions may reside on the disk or the file server. The INTERLISP func-

tions and predicates that manipulate files always retrieve the latest version of the file unless a version is explicitly specified in the argument to one of these functions. It is good practice to always keep at least one or two old versions of a file on the disk. More than that tends to clutter the disk and consume space that may otherwise be needed for data.

Unlike INTERLISP-10, [DSK] and the file servers do not support temporary files. Files that are created remain in place until you specifically delete them. However, you can simulate temporary files by defining them to reside in your virtual memory. To do so, prefix the file name with the host name [CORE]. All files defined in this manner reside within the user's virtual memory; they are saved with a SYSOUT of your virtual memory and are only lost when your virtual memory is abandoned.

16.2 FILE DECLARATION

Before a program can use a file, it must declare it to the system. A file may be declared to be input, output, or both upon its opening. After a file has been opened, the program may specify a file to be either its primary input or output file. When this occurs, all input will be accepted from or output directed to that file until a new declaration is executed.

16.2.1 The Primary File 'T'

When an INTERLISP program is initiated, it has associated with it a *primary file* for input and output operations. This file is commonly known as the file 'T' and is associated with the user's terminal. Until this primary file is changed, all input is accepted from and all output is directed to the user's terminal unless an optional file name is specified in the I/O function. For example, executing the PRINT function produces the indicated string on the user's terminal:

```
←(PRINT "Welcome to INTERLISP")
"Welcome to INTERLISP"
```

T is always considered to be open.

16.2.2 Declaring the Primary Input File

You may declare new primary files for input or output using two functions provided by INTERLISP.

To declare a new primary input file, you use **INPUT**, which takes the form

```
Function:       INPUT
                INFILE

# Arguments:    1
```

```
Arguments:      1) a file name, FILE
Value:          The current primary input file.
```

For example, to declare the card reader to be the primary input file, we can use the following expression (in INTERLISP/370):

```
←(INPUT 'rdr)
T
```

INPUT returns as its value the name of the old primary input file. It is generally a good idea to retain the name of the old primary input file (via SETQ) if you switch among several input files.

Note that (INPUT) returns the name of the current primary input file without changing its name.

```
←(INPUT)
⟨KAISLER⟩MY-INPUT.DAT;4
```

Executing (INPUT T) makes the primary input file the user terminal.

The file that you declare as the primary input file must already be open for input:

```
←(INPUT 'TEST.DAT)
FILE NOT OPEN
TEST.DAT
```

If the file is not open, an error message is printed and the primary input file is not changed.

You may specify a string as an input "file" without needing to open it. Consider the following example:

```
←(INFILE "JUNK")
T
←(INPUT)
"JUNK"
←(READ)
END OF FILE
NIL
←(INPUT)
" "
```

because the input operation has removed characters from the string. Reading against the string will cease if it encounters the characters STOP in the string.

```
←(INFILE "JUNK STOP")
"JUNK STOP"

←(READFILE)
(JUNK)
```

I do not recommend that you use the feature of the INFILE/READ functions to obtain data.

Opening a File as the Primary Input File
An alternative function, **INFILE**, opens the file for input and makes it the primary input file.

```
←(INFILE 'COMPLEX)
T

←(INPUT)
⟨KAISLER⟩COMPLEX..9
```

In INTERLISP-D, if the file will not open, INFILE generates an error message:

```
←(INFILE ⟨some file name⟩)
File Won't Open
```

and the primary input file is not changed.

A Definition for INFILE
We might define INFILE as follows:

```
(DEFINEQ
     (infile (file)
          (PROG (name)
               (COND
                    ((SETQ name (INFILEP file))
                         (OPENFILE name 'INPUT 'OLD)
                         (RETURN (INPUT name)))
                    (T
                         (ERROR "FILE NOT FOUND" file))))
     ))
```

16.2.3 Declaring the Primary Output File

The corresponding function for redeclaring the primary output file is **OUTPUT**, which takes the form

```
Function:       OUTPUT
                OUTFILE

# Arguments:    1

Arguments:      1) a file name, FILE

Value:          The current primary output file.
```

OUTPUT and OUTFILE operate in the same manner as INPUT and INFILE:

```
←(OUTPUT 'SCORES.DAT)
FILE NOT OPEN
SCORES.DAT

←(OUTFILE 'SCORES.DAT)
T

←(OUTPUT)
⟨KAISLER⟩SCORES.DAT;1
```

Note that if the file does not previously exist, INTERLISP creates the file name in its internal file list with the version number of 1. The file is also created on disk. If the file already exists, a new file with the next higher version number is created and opened for output.

A string may not be used to specify the file name for either OUTPUT or OUTFILE:

```
←(OUTFILE "JUNK")
ARG NOT LITATOM
"JUNK"
```

A Definition for OUTFILE

We might define OUTFILE as follows:

```
(DEFINEQ
     (outfile (file)
          (PROG (name)
               (COND
                    ((NOT (LITATOM file))
                         (ERROR "ARG NOT LITATOM" file)))
               (COND
                    ((SETQ name (OUTFILEP file))
                         (OPENFILE name 'OUTPUT 'NEW)
                         (RETURN (OUTPUT name)))
                    (T
                         (ERROR "FILE NOT FOUND" file))))
     ))
```

16.2.4 Testing Input/Output Files

Generally, you do not want to attempt to open a file if such an action will result in an error because the file does not exist. INTERLISP provides predicates for testing whether a file may be opened for input or output. They take the form

```
Function:       INFILEP
                OUTFILEP

# Arguments:    1

Arguments:      1) a file name, FILE

Value:          The full name of the file; otherwise, NIL.
```

INFILEP and **OUTFILEP** return the full name of the file that is the argument to the predicate if the file could be opened for input or output, respectively. For example, assume that our program required a data file named PRESIDENTS.LISPIN that contained some INTERLISP S-expressions concerning presidents. Before opening this file for input, we would like to ensure that the file actually exists. By using the INFILEP predicate, we can test whether or not INTERLISP recognizes the file:

```
←(INFILEP 'presidents.lispin)
[DSK]presidents.lispin;1
```

Note that both INFILEP and OUTFILEP return the full name of the file if it is recognized by the system. In this case, the full name form corresponds to that recognized by INTERLISP-D as the name of the file residing on the disk or file server.

In INTERLISP-10, if no version number is specified, INTERLISP seeks out the highest version number of the file and returns its full name for INFILEP and one version greater than the highest existing version for OUTFILEP.

16.2.5 File Name Recognition

INTERLISP provides a generic function for recognizing a file, **FULLNAME**, which takes the following form:

```
Function:       FULLNAME

# Arguments:    2

Arguments:      1) a file name, NAME
                2) a recognition mode, RECOG

Value:          The full name of the file.
```

FULLNAME attempts to locate a file on local disk or file server (INTERLISP-D) or in a system directory (INTERLISP-10) with the specified file

name. If the file is found, its full name is returned as the result of the function call.

There are four possible recognition modes accepted by FULLNAME: OLD, NEW, OLDEST, and OLD/NEW. OLD/NEW is interpreted to mean that if recognition fails because no old version of the file exists, then recognize the file as NEW. This mode is useful only when attempting to recognize a file for writing.

Consider the following examples, given that there are several versions of the file PRESIDENTS.LISPIN (numbered 1 to 3) residing on [DSK].

```
←(FULLNAME 'presidents.lispin OLD)
[DSK]presidents.lispin;3
```

returns the most recent of the old versions of the file.

```
←(FULLNAME 'presidents.lispin NEW)
[DSK]presidents.lispin;4
```

returns the next version of the file.

```
←(FULLNAME 'presidents.lispin OLDEST)
[DSK]presidents.lispin;1
```

returns the oldest existing version of the file.

```
←(FULLNAME 'presidents.lispin OLD/NEW)
[DSK]presidents.lispin;3
```

Note that specifying NEW as the recognition mode always returns a version that is one greater than the current oldest version of the file. This mode succeeds even if the file does not exist.

Note that the recognition modes do not have to be quoted as FULLNAME does not evaluate them. Any other mode specification results in the error message ILLEGAL ARG. The first argument must have a literal atom as its value; otherwise, the error message ARG NOT LITATOM is returned.

16.3 OPENING A FILE

Using INFILE and OUTFILE, we have seen that we can open a file for input or output, respectively, with a number of default attributes set by the system. INTERLISP provides two general functions for opening a file, OPENFILE and IOFILE, that allow you to establish the attributes of the file, and a predicate for testing if a file is open, OPENP.

16.3.1 A General File Open Function

OPENFILE takes a varying number of arguments depending on the attributes that you wish to set. Let us define the generic format for the function and then discuss each argument in detail:

```
Function:       OPENFILE

# Arguments:    2-5

Arguments:      1) an S-expression specifying a file name,
                   NAME
                2) an access mode, ACCESS
                3) a recognition mode, RECOGNITION
                4) the byte size for the file, SIZE
                5) a list of machine-dependent parameters,
                   PARMS

Value:          The full name of the file if it recognized
                and opened; otherwise NIL.
```

At a minimum, you must specify a file name and an access mode to OPENFILE; other arguments are optional whence the system will assume defaults.

Access mode may take one of four values: INPUT, OUTPUT, BOTH, or APPEND. These define the operations a user may perform upon the file. If the file is opened in APPEND mode, the system automatically positions the file pointer at the end of the file.

INPUT access mode limits operations upon the file to reading from the file. Attempts to write to the file will cause an error. OUTPUT or APPEND access mode limits operations upon the file to writing to it (including moving the file pointer). Note that OUTPUT access mode always implies that you want to create a new version of the file even if you specified a version number in the file name. BOTH allows you to both read and write to the file. If you want to read and write to a file, and preserve the contents of the version specified in the file name, you must open the file using BOTH and APPEND.

```
←(OPENFILE 'complex 'input 'old)
⟨KAISLER⟩COMPLEX..9

←(OPENFILE 'complex 'input 'new)
FILE WON'T OPEN
COMPLEX
```

The recognition modes are OLD, NEW, OLDEST, and OLD/NEW. When opening a file, INTERLISP first attempts to recognize the file (in the FULLNAME sense) using the specified recognition mode. If the file cannot be recognized for the given pair (⟨access mode⟩, ⟨recognition mode⟩), IN-

TERLISP returns the error message FILE NOT FOUND. Otherwise, the file is opened according to the access mode.

If the recognition mode is NIL, INTERLISP assumes a default based on the access mode. The current defaults are

Access Mode	Recognition Mode
INPUT	OLD
OUTPUT	NEW
BOTH	OLD/NEW
APPEND	OLD/NEW

We can see that the functions discussed in the previous sections are special cases of the execution of OPENFILE.

```
←(INFILE ⟨file name⟩)
```

is equivalent to

```
←(INPUT (OPENFILE ⟨file name⟩ 'INPUT 'OLD))
```

and

```
←(OUTFILE ⟨file name⟩)
```

is equivalent to

```
←(OUTPUT (OPENFILE ⟨file name⟩ 'OUTPUT 'NEW))
```

SIZE is used to determine the byte size in which to open the file. If it is NIL, the default value is used (for INTERLISP-10, it is 7; for INTERLISP-D, it is 8). A value of 7 results in the high-order bit of the byte being set to zero. This corresponds to the ASCII character instruction set. A byte size of 8 allows you to read all 8 bits of a standard byte; it is useful for processing EBCDIC files.

The fifth parameter, PARMS, is used to specify a list of machine-dependent parameters. Currently, two parameters are supported by all versions of INTERLISP:

WAIT	wait if the file is busy
DON'T.CHANGE.DATE	don't change the access date

INTERLISP-10 supports an additional parameter:

THAWED	open the file in the "thawed" mode

OPENFILE does not change the definition of the primary input or output files whereas INFILE and OUTFILE do.

OPENFILE is not currently available in INTERLISP/370.

Opening a File for Both Input and Output

IOFILE is a function that opens a file for both input and output. As such, it is just another shorthand version of the more general OPENFILE. It takes the form

```
Function:       IOFILE
# Arguments:    1
Argument:       1) a file name, FILE
Value:          The full name of the file if it is opened.
```

IOFILE opens an existing file for both input and output, but does not change the current primary input or output files.

```
←(IOFILE 'complex)
⟨KAISLER⟩complex..9
```

A Definition for IOFILE

We might define IOFILE as follows:

```
(DEFINEQ
     (iofile (file)
          (COND
               ((FULLNAME file)
                    (OPENFILE file 'BOTH 'OLD)))
     ))
```

IOFILE is not currently available in INTERLISP/370.

16.3.2 A Predicate for Testing Open Files

INTERLISP provides a predicate for testing if a file is already open. This test is often required in multiprocessing environments or after an error has occurred (to prevent reinitialization of the file). **OPENP** expects a file name and an access mode. It returns the full name of the file if the file is open in the specified mode, but otherwise returns NIL. Thus, if the file is not recognized, OPENP will not generate an error.

OPENP takes the following form

```
Function:       OPENP
# Arguments:    2
```

```
Arguments:      1) a file name, FILE
                2) an access mode, ACCESS

Value:          The full name of the file.
```

ACCESS may be INPUT, OUTPUT, or BOTH. If the access mode is NIL, then the file is validated for input or output, but not both.

```
←(OPENP 'presidents.lispin INPUT)
[DSK]presidents.lispin;3

←(OPENP 'presidents.lispin)
NIL
```

The latter case is a possibility if PRESIDENTS.LISPIN was opened in the BOTH access mode.

If no file name is specified as an argument to OPENP, then OPENP returns a list of the open files, excluding T which is always opened by the system. Thus, if we executed the following commands, we would see

```
←(INPUT (INFILE 'metro.lisp))
[DSK]metro.lisp;1

←(OPENP)
([DSK]metro.lisp;1)
```

16.4 GETTING AND SETTING FILE ATTRIBUTES

Most operating systems maintain a substantial amount of information about files in their directories. This information is used for different types of file management operations. INTERLISP allows the user to obtain or set this information in order to determine how to manipulate files.

GETFILEINFO is used for obtaining information about a file. It takes the following form

```
Function:       GETFILEINFO

# Arguments:    2

Arguments:      1) an S-expression specifying a file name,
                   FILE
                2) an attribute name, ATTRIBUTE

Value:          The current value of the attribute for the
                given file.
```

INTERLISP accepts the following attribute names for all implementations:

Attribute Name	*Meaning*
WRITEDATE READDATE CREATIONDATE	The data and time when the file was last written, last read, or originally created, respectively.

Consider the following example:

```
←(GETFILEINFO 'complex 'CREATIONDATE)
    "28-Jul-84 10:48:02"
```

IWRITEDATE IREADDATE ICREATIONDATE	The integer form of the above attributes.

Consider the following example:

```
←(GETFILEINFO 'complex 'IREADDATE)
    12036747520
```

BYTESIZE	The byte size of the file.

```
←(GETFILEINFO 'complex 'BYTESIZE)
    7
```

OPENBYTESIZE	The byte size with which the file was opened.
LENGTH	The byte position of the end of the file (the file does not have to be opened).

Consider the following example:

```
←(GETFILEINFO 'complex 'LENGTH)
    4522
```

ACCESS	The current access mode of the file; otherwise, NIL.

IWRITEDATE, etc. are equivalent to applying the function IDATE to the value returned by GETFILEINFO for the corresponding attribute.

```
←(GETFILEINFO 'COMPLEX 'IWRITEDATE)
12036747270
```

is equivalent to

```
←(IDATE (GETFILEINFO ⟨file name⟩ WRITEDATE))
12036747270
```

OPENBYTESIZE is provided because INTERLISP-10 allows you to open a file with a byte size different from its permanent byte size (that is, the one with which it was created). Note that INTERLISP-D assumes eight-bit bytes and creates files in this manner.

The information for all of these parameters is also stored in the file header written by INTERLISP using the File Package commands.

INTERLISP-D Attributes

The IRM [irm83] notes that INTERLISP-D distinguishes the type of the file using the following attribute:

```
TYPE                    either TEXT or BINARY
```

TEXT files are standard ASCII files while BINARY files are compiled code or files containing bit-map data for INTERLISP-D's window support functions.

INTERLISP-10 Attributes

The IRM [irm83] notes that INTERLISP-10 running under TENEX or TOPS-20 will accept additional attributes. These are given below:

```
SIZE                    The size of the file in pages.

←(GETFILEINFO 'complex 'SIZE)
2

PROTECTION              The "protection code" of the file;
                        returned as an integer value.

←(GETFILEINFO 'complex 'PROTECTION)
262080

DELETED                 T, if the file has been deleted;
                        otherwise, NIL.
```

And for TOPS-20

```
INVISIBLE               T, if the file has the invisible
                        attribute set; otherwise, NIL.

ARCHIVED                T, if the file has been archived;
                        otherwise, NIL.

OFF-LINE                T, if the file's contents are off-
                        line; otherwise, NIL.
```

Note that these attributes depend on specific features supported by the host operating system. For further explanation, you are encouraged to consult the manufacturer's operating system manuals.

16.4.1 Setting File Attributes

The corresponding function for setting the values of file attributes is **SETFILEINFO**, which takes the following form

```
Function:       SETFILEINFO

# Arguments:    3

Arguments:      1) a file name, FILE
                2) an attribute name, ATTRIBUTE
                3) a value, VALUE

Value:          T, if the attribute can be changed;
                otherwise, NIL.
```

SETFILEINFO returns T if is successful in changing the value of the file's attribute; otherwise, it returns NIL. Attribute values are defined by the host operating system. You should refer to the manufacturer's manuals to determine what values may be changed. You should also note that the host operating system may support other file attributes that are not accessible by INTERLISP. Thus, you may have to perform some modification of the file's attributes under the aegis of the host operating system's command language or utilities.

16.5 CLOSING FILES

The corresponding operation to opening a file is closing it. INTERLISP provides three functions for closing files: CLOSEF, CLOSEF?, and CLOSEALL.

16.5.1 Basic Closing Functions

The general file close function is **CLOSEF**, which takes the form

```
Function:       CLOSEF
                CLOSEF?

# Arguments:    1

Argument:       1) a file name, FILE

Value:          The full name of the file it has closed;
                otherwise, NIL.
```

If the file is successfully closed, its name is returned as the value of CLOSEF:

```
←(CLOSEF 'presidents.lispin)
[DSK]presidents.lispin;1
```

If the file is not currently open, CLOSEF returns the error FILE NOT OPEN.

CLOSEF may be given a null argument. If so, it performs the following operations:

1. It attempts to close the primary input file if it is not T.
2. Failing that, it attempts to close the primary output file if it is not T.
3. Failing that, it returns NIL.
4. If either of operations 1 or 2 succeeds, the respective primary file is reset to T.

CLOSEF? is an alternative form that closes the specified file if it is open. Otherwise, it returns NIL. That is, it does not generate an error, as CLOSEF does, if the files is not open. If it does succeed in closing the file, it returns the full name of the file as its value.

16.5.2 Closing All Files

Many applications open several files during their execution. When the execution terminates normally, good housekeeping requires that all files be closed to ensure their integrity. The function CLOSEALL provides a means for closing all open files except T and any typescript file that is recording your session. The value returned by CLOSEALL is a list of files that have been closed. It takes the following form:

```
Function:       CLOSEALL
# Arguments:    0
Arguments:      NIL
Value:          A list of files that it has closed.
```

Consider the following example:

```
←(IOFILE 'COMPLEX)
⟨KAISLER⟩COMPLEX..9
←(IOFILE 'SHK)
⟨KAISLER⟩SHK..19
```

```
←(CLOSEALL)
(⟨KAISLER⟩COMPLEX..9 ⟨KAISLER⟩SHK..19)
```

Note that certain files may be protected from the operation of CLOSEALL by the Whenclose Package, which is discussed in the next section.

16.5.3 The Whenclose Package

Closing a file is a serious matter in a program—one not to be taken lightly because it removes all knowledge of the file from the system. It can be disastrous to close a file before you have updated it with the last changes to its contents. Also, if you have written a program that others may use, you may want to perform certain housekeeping operations on the file of which they should remain ignorant. INTERLISP provides a convenient function for handling this problem, WHENCLOSE.

WHENCLOSE is a function that assigns values to certain properties associated with a file name that has been recognized by the system. These properties are inspected prior to closing the file (by any of the functions mentioned above or by a SYSOUT) to ensure that the appropriate actions are taken.

The generic format for WHENCLOSE is

```
Function:       WHENCLOSE

# Arguments:    3-11

Arguments:      1) a file name, FILE
                2) a property name, PROPERTY
                3) a value for the property, VALUE
                4-11) repeat 2 & 3 by pairs

Value:          The full name of the file.
```

WHENCLOSE is a nospread function. FILE must be the name of a file other than T (the terminal). If file is NIL, WHENCLOSE assumes you refer to the primary input and output files if they are other than T.

WHENCLOSE currently recognizes the following properties:

BEFORE

The property value specifies a function that is to be executed immediately prior to the closing of the file. The function is passed the full name of the file as its argument by INTERLISP. For example, if you are writing a file by means of WRITEFILE, before you close it you may want to write the atom STOP at the end of the file. You can specify this operation as follows:

```
←(WHENCLOSE ⟨filename⟩ 'BEFORE (FUNCTION ENDFILE))
⟨filename⟩
```

You may assign multiple BEFORE functions to a file, whence they will be executed in sequence of declaration.

AFTER
The property value specifies a function that is to be executed immediately after the file has been closed. The function is passed the full name of the file as its argument by INTERLISP.

An application program may open and close files for you without your being aware that the program has done so. Sometimes, it is useful to reassure the user when a file has been closed successfully. Because WHENCLOSE uses CLOSEF, you are assured that the file was closed successfully if CLOSEF returns the file name. Thus, you can provide a function that notifies the user that the file has been closed.

```
←(WHENCLOSE ⟨file name⟩ 'AFTER (FUNCTION notify-user))
⟨filename⟩

←(DEFINEQ
    (notify-user (filename)
        (PRIN1 filename)
        (PRIN1 " has been closed.")
        (TERPRI)
    ))
(notify-user)
```

You may assign multiple AFTER functions to a file, whence they will be executed in sequence of declaration.

STATUS
The property value specifies a function that is to be executed immediately prior to executing a SYSOUT. The value of the function is a list consisting of two parts:

CAR-portion:	The name of a function that is to be APPLY'd when the sysout file is restarted.
CDR-portion:	An S-expression to which the above function is APPLY'd to re-establish the status of the file.

Thus, the function associated with the property value is expected to create the S-expression that is to be executed when the sysout file is reloaded for execution.

INTERLISP checks the value of the APPLY'd function. If it is NIL, INTERLISP assumes that the file's status could not be restored. It prints a warning

message to the user and executes any functions associated with the file are executed.

CLOSEALL
The property value is either YES or NO. If it is YES, then this file may be closed by CLOSEALL (the function!). If it is NO, then CLOSEALL (the function!) will ignore the file. Thus, even though you use CLOSEALL to close all open files, you may protect some of them from its actions by assigning YES to this property for the respective files.

EOF
The property value specifies a function that is executed when an end-of-file condition is detected by INTERLISP. The function can determine what action to take given the condition and the state of the program.

There are two exits from the specified function. A normal exit will cause the normal error handling machinery to be invoked by INTERLISP. Alternatively, you may use RETFROM (see Section 30.8) to return from the function, whence the error handling machinery is bypassed. Note that the file will not be closed if the EOF function does not close it.

When a file is opened by INTERLISP, it is automatically initialized with the values of YES for the CLOSEALL property and CLOSEF for the EOF property.

16.6 OTHER FILE OPERATIONS

Most operating systems provide a rich set of functions for managing the file space of the user. INTERLISP provides access to the underlying operating system functions for deleting and renaming files.

16.6.1 Deleting Files

INTERLISP allows a user to delete a file by executing **DELFILE**, which takes the following form

```
Function:       DELFILE
# Arguments:    1
Argument:       1) a file name, FILE
Value:          The file name, if deleted;
                otherwise, NIL.
```

In order to properly delete files, you usually must be specific about the version to be deleted. If the version is omitted, INTERLISP attempts to delete the oldest version of the file that is recognized in the directory. To delete a specific version, you must supply the version number as part of the file name.

```
←(DELFILE 'COMPLEX)
⟨KAISLER⟩COMPLEX..6

←(DELFILE 'COMPLEX..5)
FILE NOT FOUND
COMPLEX..5
```

because I have previously deleted it.

16.6.2 Renaming Files

You may rename a file using **RENAMEFILE**, which takes the following form

```
Function:       RENAMEFILE

# Arguments:    2

Arguments:      1) the old file name, OLD
                2) the new file name, NEW

Value:          The new file name if renaming is
                successful.
```

If the version is omitted, RENAMEFILE attempts to rename the most recent version of the file found in the directory. The new file name is created with its version number set to 1. If the new file name already exists in the directory, RENAMEFILE returns NIL.

```
←(FULLNAME 'COMPLEX)
⟨KAISLER⟩COMPLEX..9

←(RENAMEFILE 'COMPLEX 'COMPLEXARITH)
⟨KAISLER⟩COMPLEXARITH..1

←(FULLNAME 'COMPLEX)
⟨KAISLER⟩COMPLEX..8
```

where version 9 of COMPLEX was renamed to COMPLEXARITH.

16.7 MANIPULATING FILE NAMES

Each operating system has its own conventions for constructing file names, and for specifying them to file management functions. A full file name, as discussed above, consists of several fields. Dynamic construction of a file name can be a tedious process involving extensive string manipulation. It is often prone to error, usually due to misplaced punctuation or field markers, that can lead to catastrophic results for your files. INTERLISP provides several functions for manipulating file names.

16.7.1 Unpacking a File Name

You may unpack a file name using **UNPACKFILENAME**, which takes the form

```
Function:       UNPACKFILENAME
# Arguments:    1
Argument:       1) a file name, FILE
Value:          A list describing the fields and values of
                the file name.
```

UNPACKFILENAME produces a list consisting of alternating *field names* and *field values*. The field values are the appropriate constituents of the given file name.

```
←(UNPACKFILENAME 'presidents.lispin)
(NAME presidents EXTENSION lispin)
```

Note that UNPACKFILENAME does not attempt to recognize the file.

The fields that may be produced by UNPACKFILENAME are HOST, DIRECTORY, NAME, EXTENSION, and VERSION in INTERLISP-D. INTERLISP-10 recognizes the additional fields of PROTECTION, ACCOUNT, and TEMPORARY.

```
←(UNPACKFILENAME '<KAISLER>COMPLEX;9)
(DIRECTORY KAISLER NAME COMPLEX VERSION 9)
```

16.7.2 Constructing File Names

You may construct a file name for submission to a file management function using **PACKFILENAME**, which takes the form

```
Function:       PACKFILENAME
# Arguments:    2-N
Arguments:      1) a field name, FIELD
                2) a field content, CONTENT
                3-N) pairs of field and content
Value:          A full file name.
```

The argument to PACKFILENAME is a list consisting of alternating field names and field values.

```
←(PACKFILENAME (LIST 'DIRECTORY 'lisp
                     'NAME ''assertions"
                     'EXTENSION 'lisp))
⟨lisp⟩assertions.lisp
```

Note that the field values may be specified as either atoms or strings.

16.7.3 Accessing a File Name Field

Given a file name, you may want to extract a specific field. **FILENAMEFIELD** allows you to extract the field value from a file name. It takes the form

```
Function:       FILENAMEFIELD
# Arguments:    2
Arguments:      1) a file name, FILE
                2) a field name, FIELD
Value:          The value of the field for the given file
                name.
```

INTERLISP recognizes the field names mentioned in Section 16.7.1. Consider the following example to retrieve the extension of a file:

```
←(SETQ a-file-name
   (PACKFILENAME (LIST 'NAME 'assertions
                       'EXTENSION 'lisp)))
assertions.lisp
←(FILENAMEFIELD a-file-name 'EXTENSION)
lisp
```

A special field value, BODY, allows you to *splice* values into the argument list to PACKFILENAME. Suppose that you merely wanted to change the directory name of a file. You could use the following form:

```
←(SETQ new-directory 'kaisler)
kaisler
←(PACKFILENAME 'DIRECTORY new-directory
               'BODY a-file-name)
⟨kaisler⟩assertions.lisp
```

If the same field name is given twice in the argument list, INTERLISP uses only the first occurrence. Suppose you wanted to provide an extension to a file only if there is no current extension in the file name. You might use the following statement:

```
←(SETQ a-file-name (LIST 'NAME 'assertions))
(NAME assertions)

←(SETQ a-default 'lispin)
lispin

←(PACKFILENAME 'BODY a-file-name
               'EXTENSION a-default)
assertions.lispin
```

This statement produces the given result because there is no extension specified for the file name.

16.8 RANDOM ACCESS FILES

INTERLISP reads and writes most files as if they were serial byte streams. In this mode, a file is read starting from the initial byte. Successive bytes are accessed in order until the last byte has been read. A file is written by sequentially placing bytes one after the other on the output stream until output is terminated. The length of the file is the number of bytes that currently reside in it.

Serial files are not very efficient when the file is very large and only portions of its contents are required for processing. To remedy the situtaion, INTERLISP provides a limited random access capability to files. Underlying this mechanism is the notion of a *file pointer*. The value of the file pointer, when valid, corresponds to a byte location in the file. By assigning different values to the file pointer, you may specify the beginning address where data will be read from or written to. Thus, random accessing allows you to jump around the file gathering data from or scattering data to different locations.

Unlike serial files, random access files require that you exercise additional caution: When you set the file pointer to the end of the file and begin to write, the file grows as succesisve bytes are placed in the output stream. It is possible to place the file pointer beyond the end of the file (by specifying a nonexistent byte address). When you commence writing, the bytes are placed at successive locations beginning with the current value of the file pointer. However, a "hole" has been created, namely the sequence of bytes between the former end of the file and the new value of the file pointer. You may later write into these locations. If you do not, errors may result when you later attempt to read from them.

Specifying a location in the file for writing will cause the new bytes to overwrite the existing bytes. Careful control of where you read and write in a file must be exercised in order to prevent the obliteration of data. A number of authors have described random access techniques in considerable detail, including [knut68] and [wied77].

In the following sections, we discuss functions that allow you to manipulate the file pointer and randomly access files.

16.8.1 Manipulating the File Pointer

You may get or set the file pointer. **GETFILEPTR** returns the current byte address in the file. The next input or output operation will begin at this location if the file pointer remains unmodified.

GETEOFPTR returns the byte address of the last byte in the file. You may think of it as determining the length of the file (remembering that the first byte's address is 0). These functions take the form

```
Function:       GETFILEPTR
                GETEOFPTR

# Arguments:    1

Argument:       1) a file name, FILE

Value:          The current or last byte address,
                respectively, of FILE.
```

If the file is not open, both functions will generate an error "FILE NOT OPEN!". Consider the following example:

```
←(OPENFILE 'JUNK 'OUTPUT 'NEW)
⟨KAISLER⟩JUNK..1

←(GETFILEPTR 'JUNK)
0
```

Now, let us write an expression to the file:

```
←(WRITEFILE (GETD 'COMPLEX) 'JUNK)
⟨KAISLER⟩JUNK..1

←(GETEOFPTR 'JUNK)
370
```

Setting the File Pointer

SETFILEPTR allows you to modify the value of the file pointer. Three cases are possible:

1. ADDRESS ⟨ 0, causes the file pointer to be set to the end of the file.
2. 0 ⟨= ADDRESS ⟨ EOF, causes the file pointer to be set to the corresponding location.
3. ADDRESS =⟩ EOF, causes the file pointer to be set to the corresponding location, but a hole is created. The file is not enlarged until you actually write data at the new location.

SETFILEPTR takes the format

```
Function:       SETFILEPTR

# Arguments:    2

Arguments:      1) a file name, FILE
                2) an address, ADDRESS

Value:          The new value of the file pointer.
```

Consider the following example:

```
←(SETFILEPTR 'JUNK 100)
100

←(READ 'JUNK)
(R I)
```

which are the arguments to the function COMPLEX that was written to the file (see Section 13.7.2 for the full definition).

```
←(READ 'JUNK)
(* edited: "17-Jul-84 21:02")

←(GETFILEPTR 'JUNK)
138
```

16.8.2 Testing for Random Accessibility

You may test whether or not a file is randomly accessible using **RANDACCESSP**, which takes the form

```
Function:       RANDACCESSP

# Arguments:    1

Argument:       1) a file name, FILE

Value:          FILE, if it is randomly accessible;
                otherwise, NIL.
```

Certain files are inherently not randomly accessible: T or NIL. Certain site-specific files (such as LPT) which correspond to psuedo-devices may also have a serial nature. FILE must be open or an error will be generated.

```
←(RANDACCESSP 'JUNK)
⟨KAISLER⟩JUNK..1

←(RANDACCESSP T)
NIL
```

16.8.3 Searching a File

You may search a file for a specified pattern using either **FILEPOS** or **FFILEPOS**, which take the form

```
Function:       FILEPOS
                FFILEPOS

# Arguments:    7

Arguments:      1) a pattern, PATTERN
                2) a file name, FILE
                3) a starting address, START
                4) an ending address, END
                5) a wild card character, SKIP
                6) a tail flag, TAIL
                7) a case array, CASE

Value:          A byte address in FILE; otherwise, NIL.
```

FILEPOS and FFILEPOS correspond to STRPOS (see Section 10.6) except that they search files rather than in-memory strings.

(F)FILEPOS searches a file from START to END. If START is NIL, the current file pointer value is used. If END is NIL, the search will proceed to the end of the file.

If the search is successful, the address of the byte sequence corresponding to PATTERN in the file is returned. If TAIL is T, the address of the first byte after the matching pattern is returned instead. If the search fails, NIL is returned. The file pointer is set to either of the byte addresses returned so that the next input or output operation may commence from that point. It is not modified if the search fails.

SKIP is a character that matches any byte in the file at the corresponding location in PATTERN.

Suppose we want to search the file JUNK for the sequence of characters (R I). We can use the following call:

```
←(OPENFILE 'JUNK 'INPUT 'NEW)
⟨KAISLER⟩JUNK..1

←(FILEPOS '(R I) 'JUNK 0 NIL)
102
```

FFILEPOS differs from FILEPOS in that it uses a different search algorithm. The IRM [6.1.4] notes that FFILEPOS will be significantly faster for larger strings. You should consult the IRM for more details.

Case Arrays

Searching may distinguish between upper- and lower-case alphameric characters or it may map several characters, which are usually ignored, into a single

character. To do so, you must supply a *case array*, an array of 128 integers. Each byte in FILE that is to be matched will be mapped through CASE, if non-NIL, before matching. Character code i in FILE becomes j via (ELT CASE (ADD1 i)). You may create a case array using **CASEARRAY**, which takes the form

```
Function:       CASEARRAY
# Arguments:    1
Arguments:      1) An old case array address, OLDARRAY
Value:          The address of a new case array.
```

CASEARRAY creates and returns the address of a new case array with all elements set to themselves to indicate the identity mapping.

```
←(SETQ case1 (CASEARRAY))
{ARRAYP}#542621
←(ELT case1 24)                    "INTERLISP-10"
23
```

because INTERLISP-10 arrays begin at 1.

Setting a Case Array

You may set an element of a case array using **SETCASEARRAY**, which takes the form

```
Function:       SETCASEARRAY
# Arguments:    3
Arguments:      1) a case array address, CASEARRAYX
                2) a character code, FROMCODE
                3) a character code, TOCODE
Value:          The value of TOCODE.
```

SETCASEARRAY inserts the value of TOCODE into the element given by FROMCODE in CASEARRAYX. Thus, when CASEARRAYX is used in a file search, all bytes having the value FROMCODE will be translated to TOCODE.

Setting Break or Separator Characters

If you do not want to match break or separator characters, **SEPRCASE** returns a case array in which all break or separator characters identified in FILERDTBL are mapped into the same character, a blank or space. SEPRCASE takes the following form

```
Function:       SEPRCASE
# Arguments:    1
Argument:       1) a clisp flag, CLISPFLG
Value:          The address of a case array.
```

The case array which is returned by SEPRCASE has all break and separator characters mapped into the character code 0. If CLISPFLG is not NIL, all CLISP characters are mapped into that same character as well.

16.8.4 Copying Bytes from File to File

You may copy bytes from one file to another using **COPYBYTES**, which takes the form

```
Function:       COPYBYTES
# Arguments:    4
Arguments:      1) a source file name, SRCFILE
                2) a destination file name, DSTFILE
                3) a starting byte address, START
                4) an ending byte address, END
Value:          T
```

COPYBYTES generates an error "FILE NOT OPEN" if either or both of SRCFILE or DSTFILE are not open.

COPYBYTES copies bytes from SRCFILE to DSTFILE beginning at START and continuing up to but not including END. If END is NIL, START is the number of bytes to be copied from SRCFILE's current file pointer. If START is also NIL, bytes are copied from the current file pointer of SRCFILE to the end the file. Consider copying bytes from JUNK to the terminal:

```
←(OPENFILE 'JUNK 'INPUT)
⟨KAISLER⟩JUNK..1

←(COPYBYTES 'JUNK T 100 138)
(R I)
(* edited: ''17-Jul-84 21:02")
T

←(COPYBYTES 'JUNK T 200 225)
ORDACCESS (QUOTE REAL)
T
```

OPYBYTES begins copying at the exact byte that you specify. Thus, if you do properly specify a valid starting byte in the source file, you may put garbage into the destination file.

16.8.5 Testing for an End of File

Before writing, you usually want to know if the file pointer is at the end of the file. **EOFP** allows you to test for an end of file. It takes the form

```
Function:       EOFP
# Arguments:    1
Argument:       1) a file name, FILE
Value:          T, if the file pointer points to the last
                byte in FILE.
```

FILE must be open (at least for input) in order to test for the end-of-file condition. Consider the following example:

```
←(EOFP 'JUNK)
NIL
←(SETFILEPTR 'JUNK -1)
-1
←(EOFP 'JUNK)
T
```

16.9 SAVING AND RESTORING A USER'S VIRTUAL MEMORY

Each INTERLISP user works with his own private memory. Because the INTERLISP address space is greater than most physical memories, a virtual memory management scheme is used. However, this private memory is treated as a large linear name space. That is, a user may access any symbol defined within the name space. Partitions of symbols into collections are defined by the user according to the functions he has built or used.

This linear name space means that, in most cases, a user may work profitably at only one problem at a time. However, many users tend to work on several problems. A linear name space prevents these different problems from having the same symbol names without conflict in meaning.

In addition, only one copy of the virtual memory file is maintained on mass storage for each user. This virtual memory file is used to temporarily store portions of a user 's program until they are referenced by the CPU. A single copy of the virtual memory file is required because INTERLISP manages that file in a manner transparent to the user except for some small performance degradations.

If you are working on multiple independent problems, each problem should be physically separate in order to avoid name space conflicts. You may do this by creating separate copies of each problem's environment (called *saving*). To rees-

tablish an evironment, you reload a copy into the INTERLISP virtual memory (called *restoring*).

16.9.1 Saving Your Virtual Memory

SYSOUT is used to save your private memory in a permanent disk file. SYSOUT takes the form

```
Function:       SYSOUT
                SYSOUTP

# Arguments:    1

Argument:       1) a file name, FILE

Value:          The value of FILE.
```

FILE specifies the disk file where a copy of your INTERLISP environment will be saved.

```
←(SYSOUT '⟨kaisler⟩aiwork.sysout)
⟨KAISLER⟩AIWORK⟩SYSOUT..1
```

will save a copy of my current INTERLISP environment (which presumes an AI flavor) in the file AIWORK.SYSOUT in my current directory.

If FILE is non-NIL, SYSOUTFILE is set to FILE. Normally, if FILE is NIL, the value of SYSOUTFILE will be used to store the virtual memory. The default value for SYSOUTFILE on INTERLISP-D is VIRTUALMEM. The default value for SYSOUTFILE in TENEX is WORK.EXE. If no extension is specified for the file name in FILE, the value of SYSOUT.EXT is used. It has the value SYSOUT for INTERLISP-D and EXE for INTERLISP-10 (TENEX).

SYSOUT does not save the state of any open files. WHENCLOSE (see Section 16.5.3) may be used to specify the operations necessary to re-establish the file environment when the sysout is reloaded at a later time.

SYSOUT returns NIL if the save operation fails for any reason; otherwise, it returns the file name of the file which was created.

Note: Failure may occur at the operating system level rather than within INTERLISP in which case SYSOUT never returns. For example, errors may result due to bad disk sectors, invalid virtual memory structure after a crash, or lack of sufficient disk space to accommodate the entire virtual memory file.

Determining if a File is a SYSOUT

You may determine whether or not a file that already exists was written by SYSOUT by executing the predicate **SYSOUTP**. SYSOUTP takes one argument—the file name to be checked. SYSOUTP returns ? if the file was previously created by SYSOUT; otherwise, NIL. SYSOUTP is defined only for INTERLISP-10.

In INTERLISP-10, SYSOUT saves only those portions of the virtual memory which you have modified. Unchanged pages are retrieved from the makesys file of the INTERLISP system. This is because many users share a single common makesys file in INTERLISP-10, which is a large, multi-user, timesharing system.

In INTERLISP-D, a sysout file contains a copy of the entire allocated memory. This is because Xerox 11xx machines are single-user workstations. For more information concerning the Xerox 11xx machines, consult the IRM.

16.9.2 Restoring Your Virtual Memory

You may restore an INTERLISP environment by executing the function **SYSIN**, which takes the form

```
Function:       SYSIN
# Arguments:    1
Argument:       1) a file name, FILE
Value:          The name of the file.
```

SYSIN copies the environment contained in the argument file into the INTERLISP virtual memory. Execution begins immediately after the SYSOUT invocation.

```
(SYSIN '⟨kaisler⟩aiwork.sysout)
```

would restore my AI working environment.

Note that SYSIN resumes execution immediately after the SYSOUT which created the permanent file. Thus, an expression of the following form would print the specified message upon return from a SYSIN:

```
(PROGN
     (SYSOUT '⟨kaisler⟩aiwork.sysout)
     (PRIN1 "Hello, Steve!")
     (TERPRI)
     (PRIN1 "Ready to begin work again, are you?")
     (TERPRI))
```

would print, after a SYSIN,

```
←(SYSIN '⟨kaisler⟩aiwork.sysout)
Hello, Steve!
Ready to begin work again, are you?
```

SYSIN is defined for INTERLISP-10 only because it is a multi-user, time-sharing system. On INTERLISP-D, you may save and restore your virtual memory from the default value of SYSOUTFILE unless multiple users share a single Xerox 11xx machine. Consult the IRM for details concerning loading sysouts under INTERLISP-D.

16.9.3 Advising SYSOUT Before Saving

SYSOUT is advised (see Chapter 21) to evaluate the S-expressions which are contained on BEFORESYSOUTFORMS. BEFORESYSOUTFORMS may contain expressions to

1. Set SYSOUTDATE to (DATE) to record the time and date that the SYSOUT was performed.
2. If the argument is NIL, then the value of SYSOUTFILE is used as the destination of the SYSOUT. After the SYSOUT, SYSOUTFILE will be set to the value of the argument. SYSOUTFILE allows a user to issue the functional form (SYSOUT) for any saves subsequent to the initial one.
3. Perform any operations associated with open files as specified by invocations of WHENCLOSE (see Section 16.5.3).
4. Any other S-expressions that you may wish to execute immediately prior to a SYSOUT necessary to record environmental information.

Note that SYSOUT does not save the status of open files in an environment. Thus, you must record their status using the WHENCLOSE package prior to the SYSOUT in order that their status may be restored when the environment is restored.

The initial value of BEFORESYSOUTFORMS is

```
((SETQ SYSOUTDATE (DATE))
 (PROGN
     (COND
          ((NULL FILE)
             (SETQ FILE SYSOUTFILE))
          (T
            (SETQ SYSOUTFILE
                  (PACKFILENAME 'VERSION NIL 'BODY
                  FILE))))
     (COND
          ((AND
               (NULL (FILENAMEFIELD FILE 'EXTENSION))
               (NULL (FILENAMEFIELD FILE 'VERSION)))
               (SETQ FILE
                     (PACKFILENAME 'BODY
```

```
                                    FILE
                                    'EXTENSION
                                    SYSOUT.EXT)))))
    (RESTOREFILES T))
```

16.9.4 Advising SYSOUT After Restoring

After a SYSIN is completed, SYSIN is advised to execute the S-expressions contained on AFTERSYSOUTFORMS. Usually, AFTERSYSOUTFORMS will contain expressions to

1. Reset the terminal line length to the appropriate value.
2. Reset any terminal control characters (using RESETFORMS).
3. If SYSOUTGAG is NIL, and the same user is executing the SYSIN as performed the SYSOUT, then the user is greeted with the value of HERALDSTRING followed by a greeting message. If the user performing the SYSIN is different, INTERLISP prints the following message to the user:

```
**** ATTENTION USER ⟨userid⟩:
THIS SYSOUT US INITIALIZED FOR USER ⟨other-
userid⟩
TO REINITIALIZE, TYPE (GREET).
```

   This allows the new user to execute his or her own personalized greeting file (see Section 29.5) to customize the environment for his or her work.

   If SYSOUTGAG is a list, the list is evaluated in lieu of printing the above messages. This allows a user to create and print his or her own greeting messages.

   If SYSOUTGAG is non-NIL and not a list (typically T), no message is printed.
4. Invokes SETINITIALS to reset the user's initials that are used for time-stamping (see Section 29.6.3).
5. Perform any operations associated with previously opened files as specified by invocations of WHENCLOSE (see Section 16.5.3).

AFTERSYSOUTFORMS has the following structure:

```
((SETQ TTYINEDIT.SCRATCH NIL)
 (RESTOREFILES)
 (COND
    ((LISTP SYSOUTGAG)
        (EVAL SYSOUTGAG))
```

```
        (SYSOUTGAG)
        ((OR
             (NULL USERNAME)
             (EQ USERNAME (USERNAME NIL T)))
             (TERPRI T)
             (PRIN1 HERALDSTRING T)
             (TERPRI T)
             (TERPRI T)
             (GREETO)
             (TERPRI T)
        (T
           (LISPXPRIN1 "****ATTENTION USER "
                       T)
           (LISPXPRIN1 (USERNAME)
                       T)
           (LISPXPRIN1 ": this sysout is initialized for
           user" T)
           (LISPXPRIN1 USERNAME
                       T)
           (LISPXPRIN1 " " T)
           (LISPXPRIN1 "To reinitialize, type GREET()"
                       T)))
    (SETINITIALS))
```

16.10 COMMENTING FUNCTION DEFINITIONS

INTERLISP can be a difficult language to read, even when it is prettyprinted. Many users, hardened by experiences with previous conventional languages, tend to choose abbreviations or names for variables, properties, and functions that are often obscure and non-mnemonic. Even when mnemonic names are chosen for user variables, properties, and functions, the wealth of system variables, properties, and functions (whose names tend to be non-mnemonic) can be quite confusing.

To make source code readable and understandable, INTERLISP provides you with a flexible commenting feature that allows you to annotate functions and files.

A *comment* is indicated by an S-expression beginning with the character *. It has the general form

```
(* ⟨comment text⟩)
```

Comments are treated like other S-expressions. Thus, * is defined as an NLAMBDA, NOSPREAD function that merely returns its argument as its value.

Comments included in function definitions are part of the definition. Comments should be placed where they cannot affect the computation. Examples include

1. Outside of COND expressions.
2. If inside a COND expression (due to complexity or length), then within one of the COND cases.
3. Outside logically complete S-expressions, particularly those that might take an indefinite number of arguments (as many File Package functions do).
4. Outside of function definitions.

Comments are intended mainly for documenting listings. When a function is prettyprinted at your terminal, comments are suppressed (presumably because you know what you are doing—not always a good idea, though!). Instead, they are replaced by the string **COMMENT**.

16.10.1 Printing Comments

PRETTYPRINT (see Section 15.7.1) provides standard treatment for comments. In many cases, this suffices to meet your documentation needs. Sometimes, you may want to emphasize a particular aspect of documentation. To do so, you need to display comments in a certain way.

You may intercept comment printing from within PRETTYPRINT in order to perform special processing on the text. To do so, you must define * as a PRETTYPRINTMACRO:

```
←(SETQ PRETTYPRINTMACROS
      '(* (LAMBDA (x)
                (RESETFORM (LINELENGTH 132)
                           (COMMENTL x))))
```

which allows you to reset the line length, print the comment, and restore the line length. The form for a PRETTYPRINTMACRO is

```
(⟨character⟩ ⟨function⟩)
```

The function is defined as a LAMBDA expression taking one argument, X, which is the text of the comment as it occurs in the S-expression. You may process the string in any manner you choose before printing it.

Comment printing is controlled by a system variable **COMMENT**FLG. If this variable is NIL, the entire text of the comment is printed by PRETTYPRINT exactly as it appears in the S-expression. If it is non-NIL, PRETTY-

PRINT substitutes its value in place of the comment text in a list. Initially, **COMMENT**FLG is set to **COMMENT**.

16.10.2 Comment Pointers

When programs are heavily commented, considerable storage may be consumed to represent the comment text. Comments are needed only for documentation and are usually manipulated during editing. If many comments appear inside a function, each is evaluated as an S-expression even though it only returns the comment text as a value. A substantial penalty is paid for setting up the system stack each time a comment S-expression is evaluated.

Rather than loading comment text into memory, INTERLISP creates a pointer to it in memory. This allows the comment to be retrieved from the file when necessary. Otherwise, it consumes a minimal amount of memory.

You may enable this feature by resetting the system variable NORMALCOMMENTSFLG to NIL. It is initially T meaning comment text is loaded into memory.

You may get the comment text from a file by using **GETCOMMENT**, which takes the form

```
Function:       GETCOMMENT
# Arguments:    2
Argument:       1) a comment pointer, COMMENTPTR
                2) a destination file, DSTFIL
Value:          The value of COMMENTPTR.
```

* is defined as a read macro (see Section 14.4) in FILERDTBL, the read table used by the File Package. When a * is encountered, the loading routines construct an expression containing

1. The name of the file.
2. The address of the first byte where the comment is located.
3. The length of the comment text.

Reading a Comment

READCOMMENT is used to implement * as a read macro. It takes the form

```
Function:       READCOMMENT
# Arguments:    3
Arguments:      1) a file name, FILE
                2) a read table, RDTBL
                3) a list, LST
```

Consult Section 14.4.5 for the definition of read macros.
* defined as a read macro appears as

```
(1OMACRO
     (X
          (AND
               (OR NCF PCF VCF)
               (COMPEM
                    (CONS (LIST '* (CAR X) ' )
                          '(COMMENT USED FOR VALUE))))
          (KWOTE (CAR X)))
```

Printing a Comment
On output, * is defined as a prettyprint macro. Its effect is to copy the comment text from one file to another. It is defined using **PRINTCOMMENT**

```
Function:       PRINTCOMMENT
# Arguments:    1
Argument:       1) a comment pointer, COMMENTPTR
Value:          The value of X.
```

Thus, on **PRETTYPRINTMACROS**, we find

```
←PRETTYPRINTMACROS
((* . PRINTCOMMENT))
```

17

The File Package

INTERLISP attempts to provide you with a wholly self-contained programming environment. To this end, it provides a comprehensive package of file management functions. This package treats a symbolic file as a database that may contain function definitions, variable values, and record declarations, among other data elements. The symbolic file serves as a repository for this information. To process it, you load the symbolic file (or portions of it) into memory. In one sense, INTERLISP provides you with the illusion of a single level of memory, although this is not fully implemented.

The File Package is a set of functions, conventions, and interfaces for facilitating the bookkeeping involved in developing and maintaining a large system comprised of many files—both code and data. The File Package is interfaced, at a lower level, to the host operating system under which INTERLISP executes. However, unlike most conventional operating systems, the File Package relieves the user of the burden of worrying about many trivial details concerning where data are located, what the structure of data is, and which files have been modified.

17.1 FEATURES OF THE FILE PACKAGE

The File Package maintains knowledge of the functions and variables contained within the file. It also will maintain information on property lists, record declarations, edit macros, hash arrays, and other INTERLISP data structures. Associated with each file is an atom whose prefix is the file name and whose suffix is COMS. For example, a file describing an augmented transition network named ATN would have an atom named ATNCOMS. The -COMS atom is a list of commands that describe how to write out the file.

The File Package provides functions for performing the following operations (which are described in the following sections)

Creating and destroying definitions of data structures associated with the file.

Methods for augmenting the command list to include new items or deleting existing items.

A method for determining whether or not an item of a given type resides on a file or any file.

A method for determining what files, if any, contain definitions for a given type of data structure.

A method for determining what files need to be updated to ensure a consistent permanent version of data structure definitions currently defined in memory.

The File Package supports three categories of operations:

1. Marking changes to files.
2. Noticing the existence of files.
3. Updating files.

17.1.1 Marking Changes to Files

Whenever an object is modified by a system function that appears in a File Package command, that function invokes MARKASCHANGED (see Section 17.3.9) to mark the object as changed. For example, modifying the definition of a function via EDITF causes that function to be marked as modified and of type FNS. The File Package needs to know what objects have been changed so it can properly update the file on disk when a MAKEFILE is issued.

The marking function's effects are not always obvious, as evidenced by the following examples:

Only those properties of an atom whose values have changed will actually be marked as changed.

Changing the value of the property EXPR of an atom really effects a change in an object that would appear in an FNS command.

Advising a function causes a change in the function's definition that results in it being marked as changed.

Modifying a variable's value by a program is not noticed by the File Package, and so must be explicitly declared.

17.1.2 Noticing Files

A file is *noticed* when it is referenced by a load function, an open function, by MAKEFILE, or when definitions are first associated with a file name via FILES? or ADDTOFILES?. The act of noticing causes the root file name to be added to FILELST.

The property FILE with value ((⟨filename⟩COMS type)) is added to the property list of the root file name. "type" indicates how the file was loaded, e.g., what function was used to access it. When FILE appears on the property list of an atom, it implies that the corresponding file has been modified since the last time it was loaded or dumped.

17.1.3 Updating Files

UPDATEFILES is invoked, periodically, to determine which files contain those items that have been changed during the session. It scans FILELST. When (or if) any files are found that contain an item that has been changed, the name of the item is added to the property FILE on the root file name. After UPDATEFILES has been executed, the files that must be dumped to ensure a permanent record of changes are just those files on FILELST whose FILE property CDR is non-NIL.

Whenever a file is (re)written by MAKEFILE, all items that have been changed are removed from the FILE property list and placed on the FILECHANGES property list. The CDR of the FILE property is set to NIL.

17.1.4 File Package Properties

The File Package keeps certain information about the status of a noticed file on the property list of the atom corresponding to the root file name. The following property names are used

FILE	When a file is noticed, the property FILE is added to the root file name's property list. It is given a value of ((⟨filecoms⟩. ⟨loadtype⟩)). ⟨filecoms⟩ is the name of the variable that contains the File Package commands for the file. ⟨loadtype⟩ indicates how the file was loaded into the INTERLISP environment (see Section 17.9).
FILECHANGES	This property contains a list of all changed items since the file was loaded. When the file is dumped, the changes in the CDR of the FILE property (see Section 17.1.3) are added to the FILECHANGES property.
FILEDATES	This property contains a list of the version numbers and corresponding file dates for this file. The version numbers and dates are used when remaking a file.

FILEMAP	This property stores a file map for the file. File maps are used to directly load individual functions from specific locations within the file. The following section discusses file maps in more detail.

Consider the following example:

```
←(LOAD 'FRAMES)
⟨KAISLER⟩FRAMES..18

←(GETPROPLIST 'FRAMES)
(FILEDATES (("11-Jul-84 19:50:48" . ⟨KAISLER⟩FRAMES..18))
FILEMAP (⟨KAISLER⟩FRAMES..18 (NIL (664 7916 (CREATE.NODE
676 . 926) (INITIALIZE.NODE 930 . 1941) ...))) FILE
((FRAMECOMS . T)))
```

where the ... indicates a lot of extraneous material.

17.1.5 File Maps

A *file map* is a data structure that describes the contents of a file. Within the filemap is a sequence of expressions that identifies each function, whether symbolically defined via DEFINE/DEFINEQ or compiled, and its beginning and ending byte addresses. These byte addresses may be used with SETFILEPTR to retrieve individual functions in the file.

Many of the File Package functions depend on the file map for correct and efficient operation. Loading selecting functions via LOADFNS is accomplished by looking up each function in the file map and retrieving it using its beginning and ending byte addresses. When you remake a file, only the functions that have been changed are written from memory to the new version of the file. Unchanged functions are merely copied from the old version to the new one.

A file map is created by MAKEFILE when it writes a new file. MAKEFILE assumes that the expressions that it is writing to the file are DEFINEQ forms. PRETTYPRINT, which actually writes the expressions to the file, must know that it is writing DEFINEQ expressions. Thus, you should never print a DEFINEQ expression onto a file yourself, but always use the FNS command (see Section 17.2.1).

File maps are written as the last expression on the file. The location of the file map is overwritten into the FILECREATED expression that appears at the beginning of the file. This allows rapid access to the file map without reading the entire file.

Two variables affect the use and creation of file maps:

BUILDMAPFLG	If BUILDMAPFLG is set to NIL, a file map will not be built by LOAD or LOADFNS, or

written by MAKEFILE. Its initial value is T.

When exporting symbolic files outside the INTERLISP environment, you may not want to include a file map. Setting BUILDMAPFLG to NIL allows you to omit these data in the file.

USEMAPFLG

If USEMAPFLG is T, the functions that normally use file maps check to see if a file map exists for the file that is to be operated upon. Three checks are performed:

1. They check to see if a file map already exists in memory.
2. If not, they check the FILECREATED expression to see if it has the address of a file map which can be read from the file.
3. If not, they build a file map, unless BUILDMAPFLG is NIL.

If USEMAPFLG is NIL, the FILEMAP property and the file will not be checked for the file map. This allows you to recover from errors or problems caused by operating upon symbolic files outside of the INTERLISP environment. The IRM suggests that editing a symbolic file that has a file map with a text editor by adding or deleting one character distorts the file map with (often) catastrophic results.

The value of USEMAPFLG is initially NIL.

17.1.6 File Package Variables

The operation of the File Package is controlled by several variables. These are described below:

FILEPKGFLG

When you enter the INTERLISP environment, all File Package operations are normally enabled. Setting FILEPKGFLG to NIL disables the

noticing of files and marking of changes. However, the File Package functions will still operate, but their results are not guaranteed to be correct.

FILELST — Its value is a list of all files (actually, their root file names) that have been noticed by the File Package.

Consider the following example:

```
←FILELST
(FRAME COMPLEX)
```

LOADEDFILELST — Its value is a list of the actual names of all files that have actually been loaded by one of the File Package loading functions (see Section 17.9). This variable is not used by the File Package, but maintained by it so that you may determine which files have been loaded.

Consider the following example:

```
←LOADEDFILELST
(⟨KAISLER⟩FRAME..19 ⟨KAISLER⟩COMPLEX.9)
```

Note that FILELST and LOADEDFILELST are not affected by the functions CLOSEF or CLOSEALL described in Chapter 16. INTERLISP distinguishes between files that have been opened for reading and writing and files that have been accessed to load functions.

Using the File Package, you may consider that you have a virtually unlimited one-level memory. Functions may be stored in disk files and loaded as they are needed by other functions or programs. Unfortunately, the virtual storage limit of INTERLISP (all versions) prevents you from deleting functions in your environment once you have loaded them. The only solution is to save everything and reinitialize the environment.

17.2 FILE PACKAGE COMMANDS

The ⟨file⟩COMS atom is a list of commands that describe how a file is to be written when the appropriate function is executed. In effect, these commands determine how to update the contents of the file which represents a (more or less) permanent version of a portion of your INTERLISP environment. A wide variety

of commands are provided that allow the user to maintain a comprehensive record of his environment between INTERLISP sessions. This section describes the form and interpretation of each File Package command.

17.2.1 Functions

You may specify which function definitions are to be written to the file by adding a function command to the ⟨file⟩COMS list. The *function* command is a list having the form

```
(FNS function[1].....function[N])
```

where "function[...]" are the names of functions whose definitions are to be written to the file. When the FNS command is interpreted, a DEFINEQ expression is written to the file for the current function definition. For example,

```
(FNS CLEANUP COMPILEFILES COMPILEFILES0 CONTINUEDIT
     LISTFILES LISTFILES1 MAKEFILE MAKEFILES ADDFILE
     ADDFILE0)
```

For each file name in an FNS command, new definitions of those functions will be written to the file when it is created. If the definition has been modified in memory after the function has been defined or loaded, that definition will be used. If the definition has not been modified, it will be copied from the existing file image on disk.

Let us assume a file of functions that operate on points. We set up the FNS command as

```
(FNS * POINTFNS)
```

and define POINTFNS by

```
(SETQ POINTFNS
      (LIST 'point.sum 'point.difference 'point.times
            'point.quotient 'point.radius
            'point.scaleby))
```

When POINT is created by MAKEFILE, a definition for POINTFNS will be written to the file as follows:

```
(RPAQQ POINTFNS
       (point.sum point.difference point.times
        point.quotient point.radius point.scaleby))
```

Each function will have a definition written to the file such that the function will be redefined when the file is loaded. The form of this definition is

```
(DEFINEQ
     (point.sum ...)
     (point.difference ...)
     .
     .
     (point.scaleby ...))
```

where ... indicates the remainder of the function's definition.

MAKEFILE (see Section 17.3.1) uses the FNS command to build the file map.

17.2.2 Variables

You may write the values of variables to a file by including a *variable* command in the ⟨file⟩COMS list. A variable command takes the following form:

```
(VARS     var[1]...var[N])
```

where the "var[...]" may have one of two formats:

1. If var[...] is an atom, then the value of the variable is written onto the file using the S-expression

   ```
   (RPAQQ var[...] value)
   ```

2. If var[...] is not an atom, it is treated as an S-expression of the form (var[...] value-definition) and it is written onto the file using the S-expression

   ```
   (RPAQ var[...] value-definition)
   ```

   Consider the following example:

```
(VARS     (FONTCHANGEFLG)
          (LISTFILESTR   (QUOTE "                    "))
          (FILELINELENGTH 72)
          (MAKEFILEREMAKEFLG T)
          (CLEANUPOPTIONS (QUOTE (LIST RC)))
          (#LISTFILESCHARS 110))
```

In each of the instances above, the value of var[...] is not an atom, but an S-expression. Thus, the value will be written to a file with an RPAQ expression. Note that the value for FONTCHANGEFLG will be NIL.

Note that if you create a shorthand form of a command, such as (FNS * POINTFNS) above, each variable containing the names of objects to be written to the file will have a VARS expression also placed in the file to re-establish its value. Thus, you should expect to see in your symbolic file an expression of the form

```
(VARS POINTFNS) (RPAQQ POINTFNS (...))
```

Suppose we are creating a database in INTERLISP for a frame application. Each node in the net is represented by an atom. When we "save" the database via MAKEFILE (see Section 17.3.1), we want to place definitions in the file that will recreate the node when the file is loaded. To do so, we define the VARS variable as (assuming a database for the USA):

```
(VARS * usavars)
```

and set its value by

```
(SETQ usavars
      (LIST 'maryland 'idaho 'virginia ...))
```

A definition for creating the value of USAVARS is written to the file as

```
(RPAQQ usavars NIL)
(ADDTOVAR usavars maryland idaho virginia ...)
```

Each variable will be defined in the file by an expression of the form

```
(RPAQ? maryland NIL)
(RPAQ? idaho NIL) and so on.
```

There are several variations on the VARS command that depend on the nature of the environment or the data.

You may wish to specify variables in the form of a compiler declaration within the file. There are three commands that allow you do so:

```
(SPECVARS vars[1] ... vars[N])
(LOCALVARS vars[1] ... vars[N])
(GLOBALVARS vars[1] ... vars[N])
```

Each of the variables associated with the commands is included in an expression of the form

```
(DECLARE: DOEVAL@COMPILE DONTCOPY var)
```

Ugly Variables

In general, you may assume that each S-expression that is printed by INTERLISP may be read by INTERLISP in the same form. Some data structures exist for which a "read" operation is not the inverse of a "print" operation. These include arrays, read tables, and user-defined data types. To handle these variables, you may include the command

```
(UGLYVARS var[1] ... var[N])
```

Suppose that you want to save the value of an array on a file so that it may later be read in. Let the name of the array be RBIS, and let the contents be expressions of the form

```
(⟨year⟩ . ⟨rbis⟩)
```

Let the array contents have the following form:

```
←(FOR I FROM 1 TO 5 (PRINT (ELT RBIS I)))
(1971 . 53)
(1972 . 55)
(1973 . 38)
(1974 . 9)
(1975 . 41)
```

To save the array in a file call RBISTATS, we need to create a variable RBISTATSCOM with the following value:

```
←(SETQ rbistatscom (LIST (LIST 'UGLYVARS 'RBIS)))
((UGLYVARS RBIS))
```

and we make the file using the following expression:

```
←(MAKEFILE 'rbistats)
⟨KAISLER⟩rbistats..1

←(PRINTDEF (READFILE 'rbistats))
((FILECREATED "25-Aug-84 10:07;13" ⟨KAISLER⟩RBISTATS..1
313 changes to:    (VARS RBISTATSCOMS RBIS))
(PRETTYCOMPRINT RBISTATSCOMS)
(RPAQQ RBISTATSCOMS ((UGLYVARS RBIS)))
(READVARS RBIS)
([Y5 0 1 (1971 . 53)
         (1972 . 55)
         (1973 . 38)
         (1974 . 9)
```

```
        (1975 . 41)
        [R5 5] ])
(DECLARE: DONTCOPY (FILEMAP (NIL))))
```

READVARS is an implementation-dependent internal function used to read special forms from a file to ensure that they are properly recreated in your INTERLISP environment.

Note that UGLYVARS does not do any checking for circular data structures. If you know that your data structures do not contain any circularities, the IRM recommends that you use UGLYVARS to print unusual data structures to files because of the savings in execution time and internal storage usage.

Horrible Variables

Some data structures are inherently circular. Attempting to print them in the normal fashion may result in infinite loops. INTERLISP provides the HPRINT package (see Section 15.1.6) to handle the printing of circular structures. To print circular structures on the file, you may include the following command:

```
(HORRIBLEVARS var[1] ... var[N])
```

which uses the HPRINT package rather than PRINT to write S-expressions on the file.

Consider the following circular data structure:

```
←(SETQ rare-gases
       (LIST 'helium 'krypton 'argon 'xenon 'radon))
(helium neon krypton argon xenon)

←(RPLACD (LAST rare-gases) rare-gases)
(helium neon krypton argon xenon ...)
```

Suppose you want to put this list on a file called ELEMENTS. We create the File Package command variable as follows:

```
←(SETQ ELEMENTSCOMS
       (LIST (LIST 'HORRIBLEVARS 'rare-gases)))
((HORRIBLEVARS rare-gases))

←(MAKEFILE 'ELEMENTS)
⟨KAISLER⟩ELEMENTS..1

←(PRINTDEF (READFILE 'ELEMENTS))
((FILECREATED "25-Aug-84 10:22:54" ⟨KAISLER⟩ELEMENTS..1
295 changes to:
         (VARS ELEMENTSCOMS RARE-GASES))
(PRETTYCOMPRINT ELEMENTSCOMS)
(RPAQQ ELEMENTSCOMS ((HORRIBLEVARS RARE-GASES)))
```

```
(READVARS RARE-GASES)
(↑ (HELIUM NEON KRYPTON ARGON XENON . [1]))
(DECLARE: DONTCOPY (FILEMAP (NIL))))
```

Initializing Variables

When you load a file into your INTERLISP environment, it may contain definitions for variables that may also have been defined in previous files. By loading the new file, you would overwrite those previous definitions with the values contained in VARS commands in the new file. To prevent this, you would have to carefully structure your file loading sequence to ensure that variable definitions are not repeated in more than one file or, if they are, do not cause damage by overwriting one another. INTERLISP provides a File Package command to assist in this process.

You may cause a variable to be defined during file loading only if the current value of the variable is NOBIND in the environment (meaning it is undefined). To do so, you include an *initialize variables* command in your ⟨file⟩COMS list. It takes the form

```
(INITVARS var[1] ... var[N])
```

Variables specified in an INITVARS command will have their values defined when a file is loaded, if and only if the variable has the value NOBIND in the environment. Thus, if the variable already has a value in the environment, its value will not be changed.

INITVARS causes an RPAQ? expression to be written on the file instead of an RPAQ expression.

17.2.3 Adding Variables to a File

Once a file is established, you may wish to add new values to those currently defined for variables in the file. To do so, you include an *add-to-variable* command in the ⟨file⟩COMS list. It has the format

```
(ADDVARS (var[1] . lst[1])
         ...
         (var[N] . lst[N]))
```

Each sublist (var . lst) writes an S-expression on the file. For each variable, if there is a value in LST that is not a member of the current value of the variable when the file is loaded, that value will be added to the value of the variable. The new value of the variable is the union of its old value from the file and the value to be added as a result of the ADDVARS expression. An example of an ADDVARS expression is

```
(ADDVARS (FILELST)
         (LOADEDFILELST)
```

```
(CFILELST)
(NOTLISTEDFILES)
(NOTCOMPILEDFILES)
(MAKEFILEFORMS
    (COND
        ((MEMB 'NOCLISP OPTIONS)
            (RESETSAVE PRETTYTRANFLG T))
        ((MEMB 'CLISP% OPTIONS)
            (RESETSAVE PRETTYTRANFLG 'BOTH)))
    (COND
        ((MEMB 'FAST OPTIONS)
            (RESETSAVE PRETTYTRANFLG NIL)))
    (COND
        ((OR
            (MEMBER 'CLISPIFY OPTIONS)
            (MEMBER 'CLISP OPTIONS))
            (RESETSAVE CLISPIFYPRETTYFLG T))
        ((OR
            (EQ FILETYPE 'CLISP)
            (MEMBER 'CLISP (LISTP FILETYPE)))
            (RESETSAVE CLISPIFYPRETTYFLG
            'CHANGES)))
    (AND
        (NEQ (LINELENGTH) FILELINELENGTH)
        (RESETSAVE (LINELENGTH
        FILELINELENGTH))))
(MAKEFILEOPTIONS    RC C LIST FAST CLISP
                    CLISPIFY NIL
                    REMAKE NEW NOCLISP CLISP% F
                    ST STF
                    (REC . RC)
                    (BREC . RC)
                    (TC . C)
                    (BC . C)
                    (TCOMPL . C)
                    (BCOMPL . C))
(NILCOMS))
```

If LST is not specified, the variable is initialized to NIL if its current value is NOBIND. This prevents errors of the form UNBOUND ATOM that might occur later in your program where you assumed the variable had been initialized to some value.

Consider the rare gases example from the last section. Let us add the gas RADON to the list of RARE-GASES. To ensure that the variable is properly updated, we add the following expression to ELEMENTSCOMS:

```
(ADDVARS (RARE-GASES . RADON))
```

You may do this as follows:

```
←(SETQ ELEMENTSCOMS
            (APPEND ELEMENTSCOMS
                    (LIST
                        (LIST 'ADDVARS
                              '(RARE-GASES . RADON)))))
((HORRIBLEVARS RARE-GASES) (ADDVARS (RARE-GASES . RADON)))

←(MAKEFILE 'ELEMENTS)
〈KAISLER〉elements..2

←(PRINTDEF (READFILE 'ELEMENTS))
((FILECREATED "25-Aug-84 10:52:05" 〈KAISLER〉ELEMENTS..2
432 changes to:
(VARS ELEMENTSCOMS RARE-GASES)
previous date: "25-Aug-84 10:22:54" 〈KAISLER〉ELEMENTS..1)
(PRETTYCOMPRINT ELEMENTSCOMS)
(RPAQQ ELEMENTSCOMS ((VARS RARE-GASES))
                    (ADDVARS (RARE-GASES RADON)))
(RPAQQ RARE-GASES (HELIUM NEON KRYPTON ARGON XENON))
(ADDTOVAR RARE-GASES RADON)
(DECLARE: DONTCOPY (FILEMAP (NIL))))

←(LOAD 'ELEMENTS)
〈KAISLER〉ELEMENTS..2
FILE CREATED 25-Aug-84 10:52:05
ELEMENTSCOMS
〈KAISLER〉ELEMENTS..2

←RARE-GASES
(radon helium neon krypton argon xenon)
```

where RADON was added to the front of the list of rare gases by the ADDTOVAR expression contained within the file.

17.2.4 Association Lists

The *association list* command allows you to specify a set of association lists. Its format is

```
(ALISTS   (alist[1] atom ...)
          ...
          (alist[N] atom ... ))
```

Each alist is the name of an association list. That is, a variable whose value is an association list. The atoms are the CARs of the association list entries. For each atom found in an association list, its definition will be written to the file if the atom is found on the association list. The expression written to the file ensures that the association list will be recreated when the file is loaded.

Consider the association list GAMES:

```
←(SETQ games
       (LIST (CONS 'bridge 'cards)
             (CONS 'baccarat 'cards)
             (CONS 'chess 'pieces)
             (CONS 'craps 'dice)
             (CONS 'scrabble 'tiles)))
((bridge . cards) (baccarat . cards) (chess . pieces)
(craps . dice) (scrabble . tiles))
```

Let us write the card games to a file called CARDGAMES. We set up the File Package command as follows:

```
←(SETQ cardgamescoms
       (LIST
          (LIST 'alists '(games bridge baccarat))))
((ALISTS (GAMES bridge baccarat))

←(MAKEFILE 'cardgames)
⟨KAISLER⟩CARDGAMES..1

←(PRINTDEF (READFILE 'cardgames))
((FILECREATED "25-Aug-84 11:04:35" ⟨KAISLER⟩CARDGAMES..1
285 changes to:
         (VARS CARDGAMESCOMS))
(PRETTYCOMPRINT CARDGAMESCOMS)
(RPAQQ CARDGAMESCOMS ((ALISTS (GAMES BRIDGE BACCARAT))))
(ADDTOVAR GAMES (BRIDGE . CARDS)
                (BACCARAT . CARDS))
(DECLARE: DONTCOPY (FILEMAP (NIL))))
```

Note that when CARDGAMES is loaded, the card games (namely, BRIDGE and BACCARAT) are added to the current value of the variable GAMES. Thus, you may incrementally build the value of a variable by loading different files.

17.2.5 Properties

You may save the values of properties of atoms by including a *property* command in the ⟨file⟩COMS list. The format of this command is

```
(PROP propname atom[1] ... atom[N])
```

A PUTPROPS expression (see Section 7.3) is written onto the file for each atom in the command that has the specified property. When the file is loaded, the effect of this command is to restore the value of the property for each atom.

If PROPNAME is a list, PUTPROPS expressions will be written for each property in the list. If PROPNAME has the value ALL, then all of the properties of each atom will be saved in the file.

Consider the following example in which we have defined several properties for states. We want to save just the GOVERNOR property of VIRGINIA, but all of the properties of MARYLAND. The File Package command is defined as follows:

```
←(PUTPROP 'virginia 'governor 'robb)
robb

←(PUTPROP 'maryland 'governor 'hughes)
hughes

←(PUTPROP 'maryland 'capitol 'annapolis)
annapolis

←(PUTPROP 'maryland 'senators '(mathias sarbanes))
(mathias sarbanes)
```

Let us save this information in a file called STATES. The File Package command is set up as follows:

```
←(SETQ statescoms
      (LIST
            (LIST 'PROP 'GOVERNOR 'VIRGINIA)
            (LIST 'PROP 'ALL 'MARYLAND)))
((PROP GOVERNOR VIRGINIA) (PROP ALL MARYLAND))

←(MAKEFILE 'STATES)
〈KAISLER〉STATES..1

←(PRINTDEF (READFILE 'STATES))
((FILECREATED "25-Aug-84 21:54:15" 〈KAISLER〉STATES..1
472 changes to:
         (VARS STATESCOMS))
(PRETTYCOMPRINT STATESCOMS)
(RPAQQ STATESCOMS ((PROP GOVERNOR VIRGINIA)
                   (PROP ALL MARYLAND)))
(PUTPROPS VIRGINIA GOVERNOR ROBB)
(PUTPROPS MARYLAND GOVERNOR HUGHES CAPITOL ANNAPOLIS
SENATORS (MATHIAS SARBANES))
(DECLARE: DONTCOPY (FILEMAP (NIL))))
```

If an atom does not have the specified property, the File Package prints a warning message "NO propname PROPERTY FOR atom". Note that if the atom has the property but its value is NIL, a PUTPROPS expression will be generated for that property.

An alternative version of the *property* command will save only the properties that actually appear on the property list of the corresponding atom. Thus, while PROP causes a NIL to be written if the value of the property for a given atom is NIL, IFPROP will produce a PUTPROPS expression only if the property explicitly appears on the atom's property list. The format of IFPROP is

```
(IFPROP propname atom[1] ... atom[N])
```

where the parameters have the same values as the PROP command.

Another version of the *property* command is PROPS. PROPS specifies pairs of atoms and properties where each atom will have its associated property written to the file via a PUTPROPS expression. The format is

```
(PROPS    (atom[1] propname[1])
          ...
          (atom[N] propname[N]))
```

where the CDR of each sublist is the property of the atom whose value is to be saved in the file. Note that each of the atoms must be a LITATOM.

17.2.6 S-expressions

You may specify arbitrary S-expressions to be written to the file using the *print* command. Its format is

```
(P expression[1] ... expression[N])
```

Each S-expression following P will be printed on the output file. Each S-expression will be evaluated when the file is loaded.

Suppose we want to initialize application program menus when a file is loaded. We may set up the print commands as follows:

```
(P
     (SETQ data.management.menu
          (create.data.management.menu))
     (SETQ database.functions.menu
          (create.database.functions.menu))
     (MASTERSCOPE 'ERASE)
     (SETQ COMMENTFONT (COPY BOLDFONT))
     ...)
```

Each of the expressions is written to the file just prior to the file map. They are the last expressions read from the file. Each expression is evaluated as it is read, and thus sets the corresponding variable.

Since these expressions are evaluated as they are read in, you may cause a program to be initiated by reading in a file. This has the effect of placing the user of the program in a specific environment governed by the program.

17.2.7 Evaluation of S-expressions

You may specify arbitrary S-expressions to be evaluated when a MAKEFILE is executed by including an *evaluation* command in the ⟨file⟩COMS list. When MAKEFILE encounters an evaluation command, it evaluates the expressions following it. This command is useful for determining whether or not certain data should be written to the file. Evaluation commands may appear anywhere in the ⟨file⟩COMS list, so they may be used to control whether or not other commands are executed based on the current state of your application. The format of an evaluation command is

```
(E expression[1] ... expression[n])
```

For example, hash arrays have an internal format that is difficult to represent as an S-expression. Rather, we would like to save expressions on the file that allow us to recreate the hash arrays when the file is loaded. We may do so using the following forms:

```
(E
     (DMPHASH CLISPARRAY)
     (DMPHASH SYSHASHARRAY))
```

which places PUTHASH expressions for creating all of the non-NIL entries of the hash array on the file.

17.2.8 Commands

You may specify arbitrary lists as commands by including a *command* command in the ⟨file⟩COMS list. Its format is

```
(COMS command[1] ... command[N])
```

where each command[...] will be interpreted as a File Package command when this command is encountered by MAKEFILE.

Here is an example of how a COMS File Package command might appear:

```
(COMS
     (FNS DO? DO?= ERRORCONTEXT WHY WHYO WHYSETUP WHYSPACE
```

```
        WHYVARS GETARG# EXPLAIN1 EXPLAINARG EXPLAINARG1
        EXPLAINARG2)
    (VARS ORDINALS (LAST?))
    (P (SETSYNTAX (QUOTE ?)
                  (QUOTE (INFIX FIRST NOESC DO?))
                  T)
       (SETSYNTAX (QUOTE ?)
                  (T EDITRDTBL))
    (ADDVARS (LISPXHISTORYMACROS (? (WHY LISPXLINE)))
             (LISPXCOMS ?))
    (USERMACROS ?=))
```

17.2.9 Comments

You may include comments in your file by including a *comment* command in the ⟨file⟩COMS list. Its format is

```
(* . text)
```

where "text" is the comment to be printed in the file.

For example, let us amend the File Package command variable STATESCOMS to read as follows:

```
((* SAVE ONLY THE GOVERNOR OF VIRGINIA)
 (PROP GOVERNOR VIRGINIA)
 (* SAVE ALL OF MARYLAND'S PROPERTIES)
 (PROP ALL MARYLAND))
```

Then, after making the file and printing it back, we see the following format:

```
←(PRINTDEF (READFILE 'STATES))
((FILECREATED "25-Aug-84 21:54:15" ⟨KAISLER⟩STATES..1
472 changes to:
          (VARS STATESCOMS))
(PRETTYCOMPRINT STATESCOMS)
(RPAQQ STATESCOMS ((* SAVE ONLY THE GOVERNOR OF VIRGINIA)
                   (PROP GOVERNOR VIRGINIA)
                   (* SAVE ALL OF MARYLAND'S PROPERTIES)
                   (PROP ALL MARYLAND)))
(* SAVE ONLY THE GOVERNOR OF VIRGINIA)
(PUTPROPS VIRGINIA GOVERNOR ROBB)
(* SAVE ALL OF MARYLAND'S PROPERTIES)
(PUTPROPS MARYLAND GOVERNOR HUGHES CAPITOL ANNAPOLIS
SENATORS (MATHIAS SARBANES))
(DECLARE: DONTCOPY (FILEMAP (NIL))))
```

Comments are ignored when the file is loaded, so they help merely to explain the contents of the file. I urge you to place as many comments as you can in your files to help both yourself and others read them.

If the first element of TEXT is a *, then MAKEFILE prints a form-feed on the file before printing the comment. This allows you to place page breaks within the file so that it is neatly formatted for later printing.

17.2.10 Advice

You may specify that functions that are "advised" when the file is created will return to that state when the file is loaded. To do so, you include an *advised* command on the ⟨file⟩COMS list. Its format is

```
(ADVISED function[1] ... function[N])
```

An appropriate expression will be written on the file to reinstate the "advised" state of the function when the file is loaded.

To reinstate the "advice" associated with the function, you may include an *advice* command in the ⟨file⟩COMS list. Its format is

```
(ADVICE function[1] ... function[N])
```

A PUTPROPS expression will be written to the file for each function that will put the advice onto its property list when the function is loaded. To reinstate the "advised" state of the function, you may execute READVISE in lieu of including advise commands.

Suppose we advise the function REAL as follows:

```
←(ADVISE 'REAL
         'BEFORE
         'FIRST
         '(PRINT "Invoking RECORDACESS"))
REAL
```

Now, let us make sure the function is really advised by asking it to work on a complex number (see Section 13.7.2).

```
←(SETQ x (COMPLEX 1.0 3.0))
"Invoking RECORDACCESS"
((1.0 . 3.0))
```

We obtain this output because we defined a DEFPRINT expression for complex numbers which is described in Section 13.7.2. In order to do the printing, we must access the REAL portion of the complex number we created, so we see that REAL has been advised.

Let us write the advised function to a file called TEST. The File Package commands appear as follows:

```
←(SETQ testcoms
      (LIST
            (LIST 'FNS 'REAL)
            (LIST 'ADVISE 'REAL)))
((FNS REAL) (ADVISE REAL))
```

Now, let us create the file TEST with the definition of REAL and its advice via

```
←(MAKEFILE 'TEST)
⟨KAISLER⟩TEST..1

←(PRINTDEF (READFILE 'TEST))
((FILECREATED "25-Aug-84 22:01:20" ⟨KAISLER⟩TEST..1
609 changes to:
          (VARS TESTCOMS)
          (ADVICE REAL))
(PRETTYCOMPRINT TESTCOMS)
(RPAQQ TESTCOMS ((FNS REAL)
                 (ADVISE REAL)
                 (ADVICE REAL)))
(DEFINEQ (REAL
         (LAMBDA (X)
               (* edited: "17-Jul-84 21:03:34")
               (RECORDACCESS (QUOTE REAL)
                             CX NIL (QUOTE FETCH)))))
(PUTPROPS REAL
          READVICE
          (NIL (BEFORE FIRST
          (PRINT "Invoking RECORDACESS"))))
(READVISE REAL)
(DECLARE: DONTCOPY
          (FILEMAP (NIL (218 384 (REAL 230 . 381)))))
```

Note that when you specify an ADVISE File Package command, INTERLISP automatically readvises the function when the file is loaded by writing a call to READVISE into the file. If you use ADVICE instead, then the advice is place on the function's property list, but you must explicitly readvise the function.

17.2.11 Macros

INTERLISP provides three types of macro definitions. The File Package provides commands for writing each type of macro definition to a file.

To save user macros, you may include a *user macro* command in the ⟨file⟩COMS list. Its format is

```
(USERMACROS atom[1] ... atom[N])
```

where each atom corresponds to a user edit macro. An S-expression is written to the file to add the edit macro definition to USERMACROS. An S-expression will also be written to place the atom name on the appropriate spelling lists.

To save "lispxmacros", you may include a *lispx macro* command in the ⟨file⟩COMS list. Its format is

```
(LISPXMACROS atom[1] ... atom[N])
```

Each atom is either a "lispxmacro" or a "lispxhistorymacro" (see Section 25.2.1). An expression will be written to the file which saves the definition of the macro so that it may be restored when the file is loaded. An expression will also be written to the file to add the macro definition to LISPXCOMS.

You may save the MACRO properties of an atom by including a *macro* command in the ⟨file⟩COMS list. Its format is

```
(MACROS atom[1] ... atom[N])
```

For each atom, an expression will be written that embeds the macro property in a form of the following type:

```
(DECLARE: EVAL@COMPILE atom)
```

where these expressions are used by the compiler.

Consider the usermacro M. We would place an entry in the ⟨file⟩COMS list to save M as follows:

```
(USERMACROS M)
```

In the file, an expression to add the definition of M to USERMACROS, the variable, would appear as

```
(ADDTOVAR USERMACROS
          (M (X . Y)
             (E (MARKASCHANGED (COND
                                 ((LISTP (QUOTE X))
                                      (CAR (QUOTE X)))
                                 (T (QUOTE X)))
                               (QUOTE USERMACROS))
                T)
             (ORIGINAL (M X . Y))))
```

17.2.12 File Package Commands

You may save user-defined file package commands or types in a file by including a *user file package* command in the ⟨file⟩COMS list. An S-expression is written to the file that restores the definition of the command or type when the file is loaded. Its format is

```
(FILEPKGCOMS atom[1] ... atom[N])
```

For example, several of the File Package commands are defined in terms of other commands. Here are several:

```
(FILEPKGCOM
     'MACROS
     'MACRO
     '(X
          (DECLARE:
               EVAL@COMPILE
               (PROPS *
                      (MAPCAR
                         (QUOTE X)
                         (FUNCTION (LAMBDA (Y)
                            (CONS Y
                                  (OR
                                      (INTERSECTION
                                      (PROPNAMES Y)
                                      MACROPROPS)
                                      (CAR
                                      MACROPROPS)))))))))
```

17.2.13 Records

You may save the declarations of record types on the file by including a *record* command in the ⟨file⟩COMS list. Its format is

```
(RECORDS record[1] ... record[N])
```

where each record [...] is the name of a record declaration. An S-expression is written to the file that will establish the record declaration when the file is loaded.

Consider the definition of a complex number whose data structure was defined as a record (see Section 13.7.2). The File Package command for saving this data structure is

```
(RECORDS COMPLEX)
```

and the expressions appearing in the file that redefines the record when the file is loaded is

```
(DECLARE: EVAL@COMPILE (DATATYPE COMPLEX ((REAL FLOATP)
                                          (IMAG FLOATP))))

(/DECLAREDATATYPE (QUOTE COMPLEX)
                  (QUOTE (FLOATP FLOATP)))
```

17.2.14 Arrays

You may save the definitions of arrays by including an *array* command in the ⟨file⟩COMS list. Its format is

```
(ARRAY variable[1] ... variable[N])
```

Each variable[...] should have an array specification as its value. An S-expression is written to the file that will re-establish the array specification when the file is loaded. this expression also resets the contents of the array to the proper values as they were when the file was written.

For example, let us define an array X and initialize it as follows:

```
←(SETQ X (ARRAY 5 0 5))
{ARRAYP}#526641

←(FOR I FROM 1 TO 5 DO (SETA X I (RAND I (PLUS I 100))))
NIL

←(FOR I FROM 1 TO 5 DO (PRINT (ELT X I)))
14
45
69
43
35
NIL
```

Now, we can save X in a file TEST by defining its File Package commands to be

```
←(SETQ TESTCOMS (LIST (LIST 'ARRAY 'X)))
((ARRAY X))
```

Then, we can make the file and inspect it as follows:

```
←(MAKEFILE 'TEST)
⟨KAISLER⟩TEST..1
```

```
←(PRINTDEF (READFILE 'TEST))
((FILECREATED "25-Aug-84 22:49:35" ⟨KAISLER⟩TEST..1
346 changes to:
          (VARS TESTCOMS X))
(PRETTYCOMPRINT TESTCOMS)
(RPAQQ TESTCOMS ((ARRAY X)))
(RPAQ X (READARRAY 5 (QUOTE 0) 1))
(14 45 69 43 35 T 5 5 5 5 5)
(DECLARE: DONTCOPY (FILEMAP (NIL))))
```

Note that READARRAY is an internal, implementation-dependent function that reads array definition data and values from a file.

17.2.15 CLISP Expressions

You may save CLISP expressions that define new iterative statement operators in a file by including a *clisp* command in the ⟨file⟩COMS list. Its format is

```
(I.S.OPRS operator[1] ... operator[N])
```

An S-expression is written to the file that redefines the value of the user-defined "i.s.opr" when the file is loaded (see Section 23.5).

17.2.16 Templates

You may save MASTERSCOPE templates in the file by including a *template* command in the ⟨file⟩COMS list. Its format is

```
(TEMPLATES atom[1] ... atom[N])
```

Each atom must have a MASTERSCOPE template (see Section 26.6). An S-expression is written to the file that restores the template when the file is loaded.

17.2.17 Blocks

You may save the definitions of blocks that are used by the compiler's block compile function in the file by including a *block* command in the ⟨file⟩COMS list. Its format is

```
(BLOCKS block[1] ... block[N])
```

For each block[...], a DECLARE expression is written to the file that will be used by the block compile functions.

17.2.18 Declarations

Expressions written to a symbolic file are either evaluated when the file is loaded, copied to the compiled file when the file is compiled (see Section 31.2), or not evaluated at compile time. You may determine when an expression is to be evaluated by including a *declaration* command in the ⟨file⟩COMS list. Its format is

```
(DECLARE: tag[1] ... tag[N]
          filepkgcom[1]
          ...
          filepkgcom[N])
```

Each tag[...] is a flag recognized by the compiler (see Chapter 31) to control evaluation. The filepkgcom[...]'s are specifications for any of the commands in this Section that describe how those commands are to be treated when a file is compiled.

17.2.19 Files

A file is loaded using the function LOAD (see Section 17.9.1). When the file is loaded, its contents are evaluated as they are read and serve to initialize your INTERLISP environment. For large programs, you may have segmented the application into many files to make it more manageable for editing and printing. Manually specifying the loading of a large number of files becomes tedious and error-prone. INTERLISP provides a File Package command that allows you to specify the automatic loading of files from within a file. It takes the form

```
(FILES file[1] ... file[2])
```

where "file" may be either a file or a list. If file[..] is an atom, it must be the name of a file, properly qualified, that is to be loaded. If it is a list, it must take the form

```
(keyword[1] ... keyword[N])
```

Several keywords may appear in a single list. Keywords allow you to conditionally modify the loading process. The currently recognized keywords are

FROM ⟨directory⟩ The file names appearing after the list are to be loaded from the specified directory. For example,

```
(FILES (FROM LISPUSERS) MULTIFILEINDEX)
```

SOURCE	The source version rather than the compiled version of the file will be loaded.
COMPILED	The compiled version of the file will be loaded. This is the default keyword. Each implementation distinguishes its compiled versions from its source versions in a different manner. INTERLISP-10 appends the extension .COM to a file that is compiled. Thus, this keyword directs the LOAD to look for files having that extension.
LOAD	Load the file using the functions LOAD? (see Section 17.9.1). This keyword loads a file only it it has not previously been loaded. It is the default mode.
LOADCOMP	Load the file using the function LOADCOMP? This keyword automatically implies SOURCE.
LOADFROM	Load the file using the function LOADFROM (see Section 17.9.4) rather than LOAD?.
SYSLOAD	Set LDFLG to SYSOUT and load the specified files. This keyword is used mainly to load system files.
PROP	Set LDFLG to PROP. This forces all function definitions to be stored under the EXPR property on property lists rather than being placed in function cells (see Chapter 8 for more detail).
ALLPROP	Set LDFLG to ALLPROP. Both function definitions and variable values will be stored on property lists rather than being stored in function cells or value cells, respectively.

17.2.20 Variations on Command Structure

The preceding sections have described a rigid format for each of the File Package commands. In each case, the arguments associated with the command must

be explicitly coded within the command. However, we do not always know what is to be saved because this is a function of the dynamic environment. The File Package provides a method for determining what is to be done during the execution of MAKEFILE.

Each of the commands previously mentioned may be followed by the atom *. If so, the expression following the * is evaluated. Its value is used to determine the arguments to the command. For example, we might say

```
(FNS makenode makeslot ...)
```

to save functions of a program on the symbolic file. But, if we create functions during the session, we would like to have those saved on the file as well. To do so, we must be able to update the parameters of the commands as we create the functions. Thus, we can define an alternative form for the FNS command as follows:

```
(FNS * ⟨filename⟩FNS)
```

where ⟨filename⟩ is the name of the file in which the function definitions are to be saved.

We would define the command list as follows:

```
(RPAQQ ⟨filename⟩COMS
       ((VARS ⟨list of variables⟩)
        (FNS * ⟨filename⟩FNS)))

(RPAQQ ⟨filename⟩FNS
       (makenode makeslot ....))
```

If the expression following the * is an atom, its value must be a list that specifies the names of the functions whose definitions are to be saved. We say that the atom is a *filevar*. The name of the atom is composed of the file name concatenated with FNS. If the expression is a list, the list is evaluated to to produce a list of function names.

Each of the commands mentioned above may utilize this alternative form of argument specification for the command. Exceptions are made for the PROP and IFPROP commands where the * must follow the property name. For example,

```
(PROP MACRO * ⟨filename⟩MACROS)
```

17.3 FILE PACKAGE FUNCTIONS

This section describes the File Package functions that are used to manipulate files. The primary function that you need is to be able to make a file that is

composed of the various INTERLISP object definitions in your environment. Other functions include listing and compiling files, and determining where (on which noticed file) an object definition resides.

17.3.1 Making Files

Whenever you modify a file, you want to ensure that the changes are reflected in the permanent copy of the file that is stored on disk. The process of creating a new file on disk is known as *making a file*. **MAKEFILE** allows you to create a new version of the file at any time, even when the file does not yet exist. The format of MAKEFILE is

```
Function:       MAKEFILE
# Arguments:    4
Arguments:      1) a filename, FILENAME
                2) an options list, OPTIONS
                3) zero or more reprint functions,
                   REPRINTFNS
                4) a source file, SOURCEFILE
Value:          The full file name (including version) of
                the new file.
```

FILENAME is the name of the file to be created. A new version of the file will always be created, even if an existing version is specified as the value of FILENAME. If FILENAME has not previously been noticed, MAKEFILE notices the file. To make a file, there must be a corresponding File Package command variable of the form ⟨filename⟩COMS defined in your INTERLISP environment. MAKEFILE uses the commands found as the value of this variable to write the proper symbolic information to the file. In most cases, PRETTYDEF (see Section 17.8.1) will be used to place symbolic information in the file.

OPTIONS is a list of arguments that direct the creation of the new file. The following options are available:

```
FAST        Forces PRETTYDEF to be invoked with
            PRETTYFLG set to NIL (see Section 17.8.1).
            The definitions of INTERLISP objects are
            printed rather than prettyprinted which
            results in a faster execution rate.

RC          Invokes RECOMPILE after PRETTYDEF to compile
            a new version of the file.

C           Invokes TCOMPL after PRETTYDEF.
```

In both cases, if the file has block definitions, BCOMPL or BRECOMPILE will be called to compile the file.

These options may be followed by a single letter from the set {F, ST, STF, S} which provides direction to the compiler (see Section 31.1).

CLISPIFY — Sets CLISPIFYPRETTYFLG to T while PRETTYDEF executes. Each function is CLISPIFYed before being prettyprinted (see Section 23.8).

NOCLISP — Sets PRETTYTRANFLG to T and calls PRETTYDEF. CLISP translations rather than CLISP expressions are printed on the file. This is useful if you are attempting to export the code to another LISP environment which does not support CLISP.

LIST — Invokes LISTFILES on FILENAME (see Section 17.3.4).

REMAKE — Prettyprints only those functions that have been changed. Functions whose definitions are unchanged from previous versions of a file are merely copied from the earlier version. In many cases, this option will cause MAKEFILE to execute faster because most sessions make very few changes in a given file in a reasonably mature application. This is the default option for MAKEFILES as specified by a value of T for MAKEFILEREMAKEFLG.

NEW — Forces a new version of the file to be created. That is, all functions are prettyprinted whether they have been changed or not.

REPRINTFNS and SOURCEFILE are used in remaking files as described in Section 17.3.2.

Consider the following example:

```
←(MAKEFILE 'SHK)
⟨KAISLER⟩SHK..1
```

where SHKCOMS was previously initialized as

```
((FNS * SHKFNS) (VARS * SHKVARS) ... other commands)
```

You cannot make a file without a File Package commands variable, e.g.,

```
←(MAKEFILE 'shksys)
SHKSYS not a file name
```

If the file was loaded from a compiled version, certain information is not present in memory to create a new symbolic file. MAKEFILE displays the warning message "CAN'T DUMP; ONLY THE COMPILED FILE HAS BEEN LOADED". This is because the noticed version of the file is the compiled code which does not contain the File Package commands to remake the file.

If only the function definitions were loaded via LOADFNS or LOADFROM, MAKEFILE displays the warning message "CAN'T DUMP: ONLY SOME OF ITS SYMBOLICS HAVE BEEN LOADED". In either case, MAKEFILE seeks your approval to perform the dump anyway. If you decline, MAKEFILE returns the value "NOT DUMPED".

A Definition for MAKEFILE

The following skeleton provides an example of the complexity of the File Package functions. The function calls are merely the "tip of the iceberg" of a powerful environmental control mechanism.

```
(DEFINEQ
        (makefile (filename options reprintfns
        sourcefile)
            (PROG (file-type x file changes file-
            dates)
                (*
                    Add filename to the list of
                    noticed files.
                )
                (addfile filename)
                (updatefiles)
                (*
                    If OPTIONS is a single atom,
                    make it a list; otherwise, do
                    nothing.
                )
                (SETQ options (MKLIST options))
                (*
                    Get the changes, if any, and the
```

```
                    current date of the file, if it
                    exists.
                )
                (SETQ changes
                      (GETPROP filename
                      'FILECHANGES))
                (SETQ file-dates
                      (LISTP (GETPROP filename
                      'FILEDATES)))
                (SETQ file-type (GETPROP filename
                'FILETYPE))
          loop0
                (COND
                    ((OR
                         (MEMBER 'NEW options)
                         (AND
                              (NULL
                              makefileremakeflg)
                              (NOT
                                (MEMBER 'REMAKE
                                options))))
                         (PRIN2 filename)
                         (PRIN1
                            (SELECTQ loadtype
                              (COMPILED
                                  "--only the
                                  compiled file has
                                  been loaded")
                              (LOADFNS
                                  "--only some of
                                  it symbolics have
                                  been loaded")
                              (HELP)))
                         (COND
                             ((NEQ
                                  (ASKUSER DWIMWAIT 'n
                                           "shall I
                                           go ahead
                                           and dump
                                           anyway?"
                                           NIL T)
                                'Y)
                                (GO out)))
                         (SETQ sourcefile NIL)
                         (SETQ reprintfns NIL))
```

```
        ((NULL file-dates)
            (*
                No FILE-DATES, so
                perhaps user made the
                COMS in memory.
            )
            (SETQ sourcefile T)
            (OR
                reprintfns
                (SETQ reprintfns
                changes)))
        ((EQUAL (filedate (CDAR file-
        dates))
                (CAAR file-dates))
            (*
                The previous version
                of the file is gone,
                so we must use the
                original and dump
                everything that has
                been changed.
            )
            (SETQ sourcefile (CDAR
            file-dates))
            (OR
                reprintfns
                (SETQ reprintfns
                changes)))

    ((AND
            (CDR file-dates)
            (EQUAL (filedate (CDADR file-
            dates))
                (CAADR file-dates)))
            (*
                The previous version
                of the file has
                disappeared, so find
                the most recent
                version and dump all
                changes.
            )
            (SETQ sourcefile (CDADR file-
            dates))
            (SETQ changes
```

```
                    (SETQ reprintfns
                       (UNION (CDR z)
                              (GETPROP
                              filename
                              'FILECHANGES)
                        ))))
              (T
                 (PRIN1   "can't find either
                          the previous version
                          or the original
                          version of")
                 (SPACES 1)
                 (PRIN2 filename)
                 (PRIN1   ", so it will have
                          to be written anew")
                 (SETQ sourcefile NIL)
                 (SETQ reprintfns NIL)
                 (SETQ options
                       (CONS 'NEW options))
                 (SETQ changes
                       (GETPROP filename
                       'FILECHANGES))
                 (GO loop0)))
        (AND
            (OR sourcefile reprintfns)
            (SELECTQ file-type
               (*
                     File was originally
                     loaded in compiled
                     form, so do a LOADFROM
                     to get declarations
               )
               (COMPILED
                  (LOADVARS
                     '((DECLARE: -- DONTCOPY
                      --))
                      (OR
                          (CDAR file-dates)
                          filename)))
               (LOADFNS
                  (*
                      For REMAKE, not all
                      variable definitions
                      will have been loaded,
```

```
                    so we must obtain
                    them.
                )
                (LOADVARS T
                        (OR (CDAR file-
                        dates)
                            filename)))
            NIL))
        (SETQ filename
            (PRETTYDEF NIL filename NIL
                    reprintfns
                    sourcefile
                    changes))
        (SETQ lastfile NIL)
        (COND
            ((NOT
               (OR
                   (EQMEMB 'DON'TLIST
                   filetype)
                   (MEMBER filename
                           NOTLISTEDFILES)))
               (SETQ NOTLISTEDFILES
                     (CONS filename
                           NOTLISTEDFILES))))
        (COND
            ((AND
                 (NOT
                   (EQMEMB 'DON'TCOMPILE
                   filetype)
                   (INFILECOMS? T 'FNS (CAAR
                   z))
                   (NOT
                     (MEMBER filename
                           NOTCOMPILEDFILES)))
                 (SETQ NOTCOMPILEDFILES
                       (CONS filename
                             NOTCOMPILEFILES))))
loop1
        (COND
            ((NULL options)
                 (RETURN filename)))
        (SETQ x options)
        (SETQ options (CDR options))
        (SELECTQ (CAR x)
```

```
                    (RC
                      (AND
                         (MEMBER filename
                         NOTCOMPILEDFILES)
                         (MAKEFILE1 filename T)))
                    (C
                      (AND
                         (MEMB filename
                         NOTCOMPILEDFILES)
                         (MAKEFILE1 filename)))
                    (LIST
                      (AND
                         (MEMBER filename
                         NOTLISTEDFILES)
                         (APPLY (FUNCTION LISTFILES)
                                (LIST filename))))
                    (OR
                      (MEMBER (CAR x)
                      MAKEFILEOPTIONS)
                      (RETURN
                         (CONS (CAR x)
                               '(- MAKEFILE
                               CONFUSED))
                         )))
               (GO loop1)
          out
               (RETURN
                    (CONS filename
                          '(NOT DUMPED))))
     ))
```

17.3.2 Remaking Files

Most symbolic files change slowly over time. That is, only a few of the function definitions change as a system is being developed. When symbolic files are large, considerable time may be consumed in creating a new version of the file. Time may be saved by rewriting only the definitions of those functions that have actually been changed. This process is called *remaking* a file. MAKEFILE operates in this mode when REMAKE is specified as an option.

You may tell MAKEFILE what is to be done. REPRINTFNS, the third argument to MAKEFILE, is a list of those functions that are to be explicitly pretty-printed on the new version of the file. SOURCEFILE, the fourth argument, specifies where the remaining definitions will be copied from. Typically, you will set both REPRINTFNS and SOURCEFILE to NIL whence MAKEFILE pretty-

prints those functions that have been changed since the last MAKEFILE or LOAD execution.

When SOURCEFILE is NIL, MAKEFILE determines the most recent version from the FILEDATES property of FILENAME. It checks to see that that version still exists. If it does not, it attempts to locate the original version of the file that was used when the session was initiated. This allows a user, during a lengthy session, to create many versions of a file via MAKEFILE without fear of losing any information in the file. Rarely does a user delete a source file after loading, so the presumption of going back to the original file is a valid one. If MAKEFILE must return to the original file, it determines which files must be prettyprinted by taking the union of all the files from the FILECHANGES property with those contained in the original file.

If MAKEFILE cannot find either the most recent version of the file or the original version, it displays the error message "CAN'T FIND EITHER THE PREVIOUS VERSION OR THE ORIGINAL VERSION OF ⟨filename⟩, SO IT WILL HAVE TO BE WRITTEN ANEW". In this case, all functions are prettyprinted.

Because function definitions may be loaded from files in several different ways (see Section 17.9), MAKEFILE must be cognizant of how these definitions were loaded. If the file was loaded from a compiled version, MAKEFILE must reload the declarations in order to write the new symbolic file. If only functions were loaded via LOADFNS, then MAKEFILE executes LOADVARS to obtain the variables as well.

17.3.3 Making Multiple Files

MAKEFILES allows you to process several files at once. MAKEFILES takes the form

```
Function:       MAKEFILES
# Arguments:    2
Arguments:      1) an option list, OPTIONS
                2) a list of files, FILES
Value:          A list of all files that are made.
```

For each file on FILES, MAKEFILES invokes MAKEFILE with OPTIONS. That is, it performs

```
(MAKEFILE (CAR files) options NIL NIL)
```

If FILES is NIL, then MAKEFILES uses FILELST. If any typed definitions have been defined or changed that do not appear on any of the files in FILELST,

MAKEFILES asks you, via ADDTOFILES?, about which files these definitions should go to.

MAKEFILES returns a list of all the files that it has made.

A Definition for MAKEFILES

We might define MAKEFILES as follows:

```
(DEFINEQ
    (makefiles (options files)
        (PROG (temporary)
              (*
                 First, ensure that all updates are
                 posted the proper files.
              )
              (updatefiles)
              (COND
                  ((NULL files)
                       (*
                          If there is a File Package
                          type object that has been
                          defined, but which has not
                          been assigned to a file,
                          ADDTOFILES? is invoked to ask
                          you where to put the object.
                       )
                       (MAPC FILEPKGTYPES
                           (FUNCTION no-resident-file))
                       (AND temporary (addtofiles?))))
              (AND
                  (ATOM options)
                  options
                  (SETQ options (LIST options)))
              (RETURN
                  (MAPCONC
                       (COND
                            ((NULL files)
                                 (*
                                     If FILES is null,
                                     use FILELST.
                                 )
                                 FILELST)
                            ((ATOM files)
                                 (LIST files))
                            (T files))
                       (FUNCTION make-each-file))))
    ))
```

The function NO-RESIDENT-FILE determines if a File Package type has been defined that is not present in any file of FILES. It will cause the printing of a warning message to the user.

```
(DEFINEQ
     (no-resident-file (type)
          (AND
               (FILES?1 type
                        (COND
                             ((NULL temporary)
                                  "Note: The following
                                  are not contained on
                                  any file:"))
                             (T (SPACES 4))))
                (SETQ temporary T))
     ))
```

MAKE-EACH-FILE determines if the name passed to it is really that of a file. If so, it invokes MAKEFILE with that name; otherwise, it merely returns NIL.

```
(DEFINEQ
     (make-each-file (filename)
          (PROG (property)
                (COND
                  ((OR
                        (NLISTP
                             (SETQ property
                                  (GETPROP filename
                                  'FILE)))
                        (NULL (CDR property)))
                        (RETURN NIL)))
                (PRIN2 filename)
                (PRIN1 '...)
                (RETURN
                    (PROG1
                         (LIST (makefile filename
                         options))
                         (TERPRI))))
     ))
```

17.3.4 Listing Files

LISTFILES displays the contents of files. It takes the following form

```
Function:       LISTFILES
# Arguments:    1
Argument:       1) A list of files to be displayed, FILES
Value:          NIL.
```

LISTFILES is an NLAMBDA, nospread function.

If FILES is NIL (the normal case), the File Package variable NOTLISTEDFILES is used. The purpose of this function is to see the files which have been changed since the last MAKEFILE. As each file name is printed, it is removed from NOTLISTEDFILES. If a file is not found, LISTFILES displays the warning "⟨filename⟩ NOT FOUND" and proceeds.

Consider the following example:

```
←(LISTFILES complex)
file ⟨KAISLER⟩COMPLEX..1 sent to the penguin
NIL
```

The IRM notes that the implementation of LISTFILES depends on the underlying operating system. INTERLISP-10 uses the printing utility of the TENEX operating system, while INTERLISP-D uses the EMPRESS utility.

17.3.5 Compiling Files

COMPILEFILES executes the RC option of MAKEFILE for each member of its argument, FILES. It takes the form

```
Function:       COMPILEFILES
# Arguments:    1-N
Arguments:      1-N) a list of filenames, FILE[1] ...
                     FILE[N]
Value:          NIL.
```

COMPILEFILES is an NLAMBDA, nospread function.

If FILES is NIL, NOTCOMPILEDFILES is used. The purpose of this function is to allow a user to make changes in a number of files and then recompile them all at once. When multiple files are used in a program and changes have to be made in several files, this provides a convenient method for ensuring recompilation of all the necessary files.

Consider the following example:

```
←(COMPILEFILES complex)
compiling COMPLEX
```

```
COMPLEX.COM not found, shall I TCOMPL <KAISLER>COMPLEX..1
instead? ... Yes

listing? ...File only
(COMPLEX (R I) NIL)
(REAL (CX) NIL)
(IMAG (CX) NIL)
(CPLUS (CX1 CX2) NIL)
(CDIFFERENCE (CX1 CX2) NIL)
(CZERO NIL NIL)
(CMULT (CX1 CX2) NIL)
(PRINT.COMPLEX (CX1) NIL)
NIL

←(COMPILEFILES test)
compiling TEST
FILE NOT FOUND
TEST
(OPENF broken)
:
```

If COMPILEFILES cannot open the specified file, it forces an error and enters the Break Package (see Chapter 20).

17.3.6 Cleaning Up Files

CLEANUP is a general workhorse that performs a MAKEFILE, LISTFILES, and COMPILEFILES for any files that appear on its argument, FILES. It takes the form

```
Function:       CLEANUP

# Arguments:    1-N

Arguments:      1-N) a list of filenames, FILE[1] ...
                     FILE[N]

Value:          NIL.
```

CLEANUP is an NLAMBDA, nospread function.

If FILES is NIL, FILELST is used. It returns a value of NIL. CLEANUP is driven by its own options list, CLEANUPOPTIONS. Initially, this variable has as its value (LIST RC). You may change the value to direct its execution.

If a file is not loaded, CLEANUP does nothing. For example,

```
←(CLEANUP complex)
NIL
```

```
←(LOAD 'COMPLEX)
⟨KAISLER⟩COMPLEX..1
FILE CREATED 28-Jul-84 10:48:03
COMPLEXCOMS

←(CLEANUP complex)
COMPLEX...file ⟨KAISLER⟩COMPLEX..2 sent to the penguin

compiling ⟨KAISLER⟩COMPLEX..2

COMPLEX.COM not found, shall I compile
⟨KAISLER⟩COMPLEX..2 instead? ... Yes
(COMPLEX (R I) NIL)
(REAL (CX) NIL)
(IMAG (CX) NIL)
(CPLUS (CX1 CX2) NIL)
(CDIFFERENCE (CX1 CX2) NIL)
(CZERO NIL NIL)
(CMULT (CX1 CX2) NIL)
(PRINT.COMPLEX (CX1) NIL)
NIL
```

17.3.7 Determining File Status

After you have been working on a program for awhile, you may want to determine the status of the files that you have been working with. **FILES?** displays on the terminal the names of those files that

Have been modified, but not dumped

Dumped, but not listed

Dumped, but not compiled

The names of any functions and types that have been defined but are not contained in any file.

It takes the following form

Function:	FILES?
# Arguments:	0
Arguments:	N/A
Value:	A list of functions and variables which have not been assigned to files and a list of files that have not been made (see example below).

If there are any functions or types that have not been assigned to files, FILES? invokes ADDTOFILES? to allow you to specify where these functions and types are to be assigned.

Consider the following example:

```
←(RECORD test (a b c d e))
TEST

←(DEFINEQ (ADD3 (a b c) (IPLUS a b c)))
(ADD3)

←(FILES?)
the records: TEST...to be dumped
the functions: ADD3...to be dumped
want to say where the above go? ... No
NIL
```

INTERLISP queries you about where the files should go via ASKUSER. If you fail to respond within the specified period, it automatically assumes NIL and exits.

A Definition for FILES?

We might define FILES? as follows:

```
(DEFINEQ
    (files? NIL
        (*
            Make sure all updates are posted to the
            files.
        )
        (updatefiles)
        (PROG (files temporary)
            (SETQ files
                (MAPCONC filelst
                        (FUNCTION files-1?)))
            (COND
                (files
                    (*
                        If there are any files that
                        have been changed, notify
                        the user of the action to
                        be performed for each file.
                    )
                    (MAPRINT files T NIL
                            "... to be dumped."
                            ', NIL T)))
```

```
                    (*
                        Inspect all objects to see if there
                        are any which have not been assigned
                        to files.
                    )
                    (MAPC FILEPKGTYPES (FUNCTION files-2?))
                    (AND temporary (ADDTOFILES?))
                    (AND
                         (SETQ files NOTLISTEDFILES)
                         (MAPRINT files T NIL
                                 "... to be listed."
                                 ', NIL T))
                    (AND
                         (SETQ files NOTCOMPILEDFILES)
                         (MAPRINT files T NIL
                                 "... to be compiled."
                                 ', NIL T))
                    (RETURN))
          ))
```

FILES-1? and FILES-2? are utility functions defined as follows:

```
(DEFINEQ
     (files-1? (x)
          (AND
               (CDR (GETPROP x 'FILE))
               (LIST x))
     ))
(DEFINEQ
     (files-2? (type)
          (AND
               (files?1 type (AND files "    plus"))
                        (AND (NULL files)
                             "... to be dumped.")))
               (SETQ temporary T))
     ))
```

FILES?1 handles the printing chores for examining FILEPKGTYPES. It may be defined as

```
(DEFINEQ
   (files?1 (type x y)
      (PROG (a-string a-list)
            (COND
```

```
                ((NOT
                   (AND
                      (LITATOM type)
                      (SETQ a-string
                            (fetch DESCRIPTION of type))
                      (LISTP
                          (SETQ a-list
                                (fetch CHANGED of
                                type)))))
                   (RETURN NIL)))
               (AND x (PRIN1 x T))
               (PRIN1 "the " T)
               (PRIN1 a-string T)
               (COND
                  (NIL
                      (IGREATERP (LENGTH a-list) 6)
                      (PRIN1 " on " T)
                      (PRIN2 (CAR type) T))
                  (T
                      (MAPRINT a-list T ": "
                               NIL ',
                               (FUNCTION files?1-indent))))
               (AND y (PRIN1 y T))
               (TERPRI T)
               (RETURN T))
))
```

where FILES?1-INDENT is defined as

```
(DEFINEQ
     (files?1-indent (x)
          (COND
               ((NOT
                    (ILESSP (IPLUS (POSITION T)
                                   (NCHARS x T T)
                                   3)
                            (LINELENGTH)))
                    (TERPRI)
                    (SPACES 8)))
          (PRIN2 x)
          ))
```

17.3.8 WHEREIS: Finding Types in Files

WHEREIS allows you to locate an instance of a file package type in a list of files that you specify. WHEREIS has the following format

```
Function:       WHEREIS

# Arguments:    3

Arguments:      1) the name of an object, NAME
                2) the type of the object, TYPE
                3) A list of files, FILES

Value:          A list of all files that contain NAME as a
                TYPE.
```

WHEREIS looks at all the files in FILES to see if the object NAME is included in the file with specified TYPE. If TYPE is NIL, WHEREIS assumes the type to be FNS, i.e., looking for a function list. If FILES is NIL, FILELST is used. Since FILELST is a list of all files "noticed" by the user as a result of loading or making files, the search may require considerable amounts of time.

Consider the following examples:

```
←(WHEREIS 'CMULT)
(COMPLEX)

←(WHEREIS 'JUNK)
NIL

←(WHEREIS 'CREATE.NODE 'FNS)
(FRAME)
```

If an object of the specified type cannot be located in any of the specified files, WHEREIS returns NIL.

17.3.9 Marking Changes in a File

Whenever a function is defined or modified, **MARKASCHANGED** is invoked to keep track of the change. This works as well for records and other file package types. Its purpose is to keep a consistent record of all modifications to a file so that the file may be written properly at the end of a session. The object is to ensure that you never lose work that you have performed during the session.

The format of MARKASCHANGED is

```
Function:       MARKASCHANGED

# Arguments:    3

Arguments:      1) the name of an object, NAME
                2) the type of the object, TYPE
                3) a reason for the change, REASON

Value:          The name of the object.
```

MARKASCHANGED marks NAME of type TYPE as being changed. REASON is a literal atom that specifies how the object was changed. Currently, MARKASCHANGED recognizes the following values:

DEFINED	An object of type TYPE called NAME has been defined in your virtual memory.
CHANGED	An object of type TYPE called NAME has been modified via the Editor.
DELETED	An object of type TYPE called NAME has been deleted by DELDEF.
CLISP	An object of type TYPE called NAME has been modified by a CLISP translation.

T is interpreted in the same way as DEFINED, while NIL is interpreted as CHANGED. Earlier versions of INTERLISP did not accept reasons, but merely noted whether the object was defined or changed.

Changes to objects are recorded in the Masterscope (see Chapter 26) database.

Consider the following example:

```
←(MARKASCHANGED 'ADD3 'FNS 'CHANGED)
ADD3
```

You may call MARKASCHANGED directly to record changes to objects. Generally, however, you will use MARKASCHANGED to record changes for user-defined datatypes.

A corresponding function, **UNMARKASCHANGED**, allows you to undo the effects of MARKASCHANGED. It takes the form

Function:	UNMARKASCHANGED
# Arguments:	2
Arguments:	1) the name of an object, NAME 2) the type of the object, TYPE
Value:	The name of the object if it was marked as changed and is now unmarked; otherwise, NIL.

UNMARKASCHANGED erases the entry in the Masterscope database that marked the object NAME of type TYPE. If no NAME of type TYPE was marked, UNMARKASCHANGED returns NIL, otherwise NAME.

```
←(UNMARKASCHANGED 'ADD3 'FNS)
ADD3
```

17.3.10 Determining What has been Changed

After you have made many changes to a file or have not modified a file for a long period of time, it is useful to find out what changes have been made. The File Package function **FILEPKGCHANGES** prints the contents of the "changes to:" section of the file header. It takes the following form:

```
Function:       FILEPKGCHANGES
# Arguments:    2
Arguments:      1) a File Package type, TYPE
                2) a list of objects, LST
Value:          A list of objects that have been marked as
                changed.
```

FILEPKGCHANGES is a LAMBDA, nospread function.

It examines all objects that have been changed (i.e., marked) but which have not yet been associated with their corresponding files. If LST is specified, the value of FILEPKGCHANGES is assigned as the value of that list. FILEPKGCHANGES returns a list of all objects marked as changed as a list of the form (typename . changedobjects).

```
←(FILEPKGCHANGES)
((RECORDS TEST) (VARS X) (FNS ADD3))

←(FILEKGCHANGES 'FNS)
(ADD3)
```

17.3.11 Adding to Files

ADDTOFILES? is a general housekeeping function. After performing a MAKEFILES, CLEANUP, OR FILES?, there may remain objects that are, as yet, unassociated with files. Often, this happens during a particularly productive programming session. In order to ensure that all these changes and creations are saved, you have to identify what files they are associated with.

ADDTOFILES? takes the following form:

```
Function:       ADDTOFILES?
# Arguments:    0
Arguments:      N/A
Value:          NIL, but has side effects (see below).
```

ADDTOFILES? asks you about the various changed, but unassociated, items, concerning what files they should belong to. You may give one of seven responses to each question (which is actually the name of an object):

1. A file name or the name of a list, such as SHOWGRAPH or SHOWGRAPHCOMS, whence the File Package uses ADDTOFILE to add the item to the corresponding file or list.
2. A ⟨line-feed⟩ indicating that the previous response should be used again.
3. A ⟨space⟩ or ⟨carriage-return⟩ indicating that no action should be taken for this item.
4. A] (⟨right-square-bracket⟩) that indicates that the item is a dummy. The item is added to NILCOMS in the file.
5. A [(⟨left-square-bracket⟩) which indicates that the definition of the item is to be prettyprinted to the terminal and the question reasked. If you had forgotten about the value of the item, this allows you to review it before deciding where it should go.
6. A (whence you are prompted with "LISTNAME: (". You must enter the name of a File Package command list. If such a list is found in the -COMS definition of the file, then the item and its definition will be added to the appropriate list. Otherwise, an error message will be printed and you will be reprompted.
7. An @ (⟨at-sign⟩) whence you are prompted with "NEAR: (". You must type in the name of another object. The specified item is inserted into the command list associated with that object. Thus, if you don't want to inspect the definition of an item but know that it is similar to one previously associated, you may simply direct the File Package to place it with the former item.

Consider the following example (assuming some unassociated objects):

```
←(ADDTOFILES?)
want to say where the above go? Yes
(variables)
X    Nowhere          "because I typed ]"
(records)
TEST File Name: COMPLEX
(functions)
ADD3 (DEFINEQ         "because I typed ["
(ADD3
     (LAMBDA (A B C) **COMMENT**
          (IPLUS A B C))))
ADD3 List: (TESTCOMS
  new list? Yes
```

```
  put list TESTCOMS on file:
(NIL)

←testcoms
(ADD3)
```

17.4 DEFINING NEW FILE PACKAGE TYPES

The previous section described a set of functions for manipulating the definitions of objects in a file regardless of their type. However, INTERLISP allows you to manipulate individual objects within a file. To manipulate new objects, you merely define how they are treated by the type-independent functions described in the following section.

17.4.1 FILEPKGTYPE

FILEPKGTYPE allows you to create new file package types. You will use it primarily to define commands that store user-defined data structures on a file. It takes the form

Function:	FILEPKGTYPE
# Arguments:	1-N
Arguments:	1) a type name, TYPE 2) a property, PROP 3) a value, VALUE 4-N) property and value pairs
Value:	The type name.

FILEPKGTYPE is a LAMBDA, nospread function.

TYPE is the name of the new file package type. PROP and VALUE have the meanings described below:

GETDEF	Defines a function that is used to retrieve the definition of an object from the file. GETDEF (see Section 17.5.1) uses this function. It takes three arguments: NAME, TYPE, and OPTIONS.
PUTDEF	Defines a function that is used to store the definition of an object on a file. PUTDEF (see Section 17.5.2) uses this function. It takes three arguments: NAME, TYPE, and DEFINITION.

DELDEF — Defines a function that is used to remove the definition of an object from a file. DELDEF (see Section 17.5.4) uses this function. It takes two arguments: NAME and TYPE.

NEWCOM — Defines a function that specifies how to make a file package command to dump an object of a given type. This function is used by ADDTOFILE and SHOWDEF. It takes four arguments: NAME, TYPE, LISTNAME, and FILE.

If this property is not specified, DEFAULTMAKENEWCOM is called to construct an expression of the form

(⟨type⟩ * ⟨filevar⟩)

where ⟨filevar⟩ either has the value of LISTNAME, if non-NIL, or the value of (FILECOMS ⟨file⟩).

WHENCHANGED — Its value is a list of functions to be applied when an object NAME of TYPE is changed or defined. Each function takes three arguments: NAME, TYPE, and NEWFLG.

```
(FILEPKGTYPE 'I.S.OPRS
             'DESCRIPTION "iterative statement
             operators"
             'WHENCHANGED '(CLEARCLISPARRAY))
```

WHENFILED — Its value is a list of functions to be applied to an object NAME of TYPE when it is added to a file. Each function takes three arguments: NAME, TYPE, and FILE.

WHENUNFILED — Its value is a list of functions to be applied to an object NAME of TYPE when NAME is removed from a file. Each function takes three arguments: NAME, TYPE, and FILE.

DESCRIPTION — Its value is a string that describes what instances of this type are.

```
(FILEPKGTYPE 'USERMACROS
 'DESCRIPTION "edit macros")
```

Where VALUE defines a function, the function may be defined as LAMBDA or NLAMBDA expression, as a FUNCTION expression, or as a list with the file name as its sole element.

You may determine the current value of a property by executing the expression:

```
(FILEPKGTYPE ⟨type⟩ ⟨property⟩)
```

You may determine the defined properties for a given type using the expression:

```
(FILEPKGTYPE ⟨type⟩)
```

Consider the following example:

```
←(FILEPKGTYPE 'FNS)
(DESCRIPTION "functions")
←(FILEPKGTYPE 'ALISTS)
(DESCRIPTION "alist entries" WHENCHANGED
(ALISTS.WHENCHANGED))
```

17.4.2 File Package Type Definitions

In this section, we give a few examples from the File Package to demonstrate how it uses the command FILEPKGTYPE to define some of the file package types that are described in Section 17.2.

```
(FILEPKGTYPE 'PROPS
             'DESCRIPTION "property lists"
             'WHENCHANGED (FUNCTION PROPS.WHENCHANGED))
```

where PROPS.WHENCHANGED might be defined as follows

```
(DEFINEQ
     (props.whenchanged (name type reason)
        (PROG (property-type)
            (SETQ property-type
                  (GETPROP (CADR name) 'PROPTYPE))
            (COND
                (property-type
```

```
                    (MARKASCHANGED (CAR name)
                                   property-type
                                   reason))
                (T
                     (SELECTQ (CADR name)
                       (CLISPWORD
                           (CLEARCLISPARRAY (CAR name)))
                       NIL))))
    ))
(FILEPKGTYPE 'EXPRESSIONS
             'DESCRIPTION "expressions"
             'WHENCHANGED (FUNCTION
             EXPRESSIONS.WHENCHANGED)
             'EDITDEF (FUNCTION NILL))
```

where **EXPRESSIONS.WHENCHANGED** might be defined as follows

```
(DEFINEQ
   (expressions.whenchanged (expression)
      (SELECTQ (CAR expression)
          ((SETQ SETQQ)
               (UNMARKASCHANGED (CADR expression)
                                'VARS))
          ((PROGN PROG)
             (MAPC (CDR expression)
                   (FUNCTION
                      (LAMBDA (x)
                         (EXPRESSIONS.WHENCHANGED x)))))
          (ADVISE
             (AND
               (EQUAL (CAADR expression)
                      (QUOTE QUOTE))
               (MAPC (PACK-IN (CADR (CADR expression)))
                     (FUNCTION
                        (LAMBDA (fn)
                           (UNMARKASCHANGED fn
                           'ADVICE))))))
          NIL))
))
```

17.5 MANIPULATING FILE PACKAGE TYPES

As we have seen, the File Package provides a powerful and flexible environment for managing your applications. The structure of a file created using the File

Package may become very complex if you take advantage of all the facilities provided by it. Consequently, the File Package inlcudes many functions to manage the type definitions that describe the environemnt and processing of a file.

Conventions

The File Package functions observe several conventions concerning the nature and processing of arguments:

1. When an ⟨atom⟩ is given where a list is expected, the functions convert the argument to (LIST ⟨atom⟩). This allows processing to proceed without causing errors on trivial mistakes (one interpretation) and makes the File Package rather user-friendly (second interpretation).
2. Singular forms of file package types are also recognized, e.g.,

VAR	VARS
RECORD	RECORDS
COM	COMS
FILE	FILES

3. If TYPE (see below) has the value NIL, a default of FNS is assumed.
4. If FILES (see below) has the value NIL, the function uses the value of FILELST (a list of all noticed files) instead.
5. The location of the definition, SOURCE, will be interpreted as follows:

?	Use the definition determined by: a. that currently in effect b. a saved definition c. a definition read from a file that is identified by WHEREIS (see Section 17.3.8) in that order.
CURRENT	Use the definition currently in effect.
SAVED	Use the saved definition (i.e., one that was stored by SAVEDEF).
FILE	Use the definition found in the first file that is identified by WHEREIS.
⟨file⟩	Use the definition, if any, that may be found in the specified file. ⟨file⟩ may be a list of files which are searched in order of occurrence in the list.
NIL	Equivalent to ?.

Note that NIL subsumes the values CURRENT, SAVED, and FILE in the order indicated. These alternatives exist to allow you to force a specific instance of a definition.

6. All functions that make destructive changes to the contents of the file are undoable.

17.5.1 Getting a Type Definition

You may obtain the definition of an item associated with a particular type using **GETDEF**, which takes the format

```
Function:       GETDEF
# Arguments:    4
Arguments:      1) an item, NAME
                2) a type idntifier, TYPE
                3) the location of the definition, SOURCE
                4) an option list, OPTIONS
Value:          An S-expression that is the definition of
                the item.
```

GETDEF reads an S-expression from SOURCE for NAME of the given TYPE. There are several cases:

FNS
If TYPE is FNS, GETDEF returns a LAMBDA/NLAMBDA expression, e.g.,

```
←(OPENFILE 'AMISMAP 'INPUT 'OLD)
[DSK]AMISMAP!5
←(GETDEF 'POINT.SUM 'FNS)
(LAMBDA (point1 point2)
    (point.new
        (PLUS
            (fetch XCOORD of point1)
            (fetch XCOORD of point2))
        (PLUS
            (fetch YCOORD of point1)
            (fetch YCOORD of point2))))
```

where with SOURCE = NIL and no current saved definition, WHEREIS has identified AMISMAP as the source of the definition. AMISMAP was noticed when it was opened.

Alternately, we can just get the arguments list via

```
←(GETDEF 'POINT.SUM 'FNS 'AMISMAP '(FAST ARGLIST))
(LAMBDA (point1 point2))
```

VARS
If TYPE is VARS, GETDEF returns the value of NAME, e.g.,

```
←(GETDEF 'RADIANS.PER.DEGREE 'VARS)
.01745328
```

RECORDS
If TYPE is RECORDS, GETDEF returns the structure of the record, e.g.,

```
←(GETDEF 'RECTANGLE 'RECORDS)
(RECORD RCETANGLE (ORIGIN . CORNER))
```

FIELDS
If TYPE is FIELDS, GETDEF returns a list of all record declarations that contain NAME as a field, e.g.,

```
←(GETDEF 'ORIGIN 'FIELDS)
(RECORD RECTANGLE (ORIGIN . CORNER))
```

FILES
If TYPE is FILES, GETDEF returns the command list for the file whose name is the value of NAME, e.g.,

```
←(GETDEF 'AMISMAP 'FILES)
(((FNS * AMISMAPFNS)
  (VARS * AMISMAPVARS)
  (RECORDS RECTANGLE)
  (CONSTANTS (PI 3.15159))
  (INITVARS
     (RADIANS.PER.DEGREE
          (QUOTIENT PI 180.0))
     ...)
  (("3-NOV-83 05:17:38" . [DSK]AMISMAP.;1))
))
```

Other Types
For all other types, GETDEF returns the S-expression that would be printed if NAME were dumped to the file as TYPE, e.g.,

```
(GETDEF 'EDITHIST 'SPECVARS 'DEDIT)
```

Otherwise
Otherwise, if GETDEF cannot find a definition for NAME, an error is caused, e.g.,

```
←(GETDEF 'POINT.MULTIPLY)
POINT.MULTIPLY
(FNS definition not found)
```

OPTIONS is an atom or a list of atoms that direct the function to adhere to certain restrictions. These include:

NOCOPY	Does not return a new copy of the definition.
NODWIM	Does not DWIMIFY (see Section 22.5) the definition.
NOERROR	Returns NIL if no definition is found.
⟨string⟩	If OPTIONS is or contains a string, it is returned when a definition for a name is not found.

17.5.2 Creating a Definition

You may create a definition for an item according to a given file package type using **PUTDEF**, which takes the form

Function:	PUTDEF
# Arguments:	3
Arguments:	1) an item, NAME 2) a type identifier, TYPE 3) a definition, DEFINITION
Value:	NAME, if successful.

PUTDEF associates DEFINITION with NAME according to TYPE. There are several cases:

FNS
If TYPE is FNS, PUTDEF executes a DEFINE, i.e.,

```
←(PUTDEF 'POINT.SUM
         'FNS
         '(LAMBDA (point1 point2)
```

```
                    (point.new
                          (PLUS point1:xcoord point2:xcoord)
                          (PLUS point1:ycoord
point2:ycoord))))
POINT.SUM
←
```

is equivalent to (DEFINE 'point.sum ...).

VARS
If TYPE is VARS, PUTDEF executes a SAVESET (see Section 25.4.1), i.e.,

```
←(PUTDEF 'x 'VARS (LIST 'a 'b 'c))
X
```

is equivalent to

```
←(SAVESET 'x (LIST 'a 'b 'c) T)
(A B C)
```

Note that using SAVESET makes the PUTDEF operation undoable (see Section 25.4).

FILES
If TYPE is FILES, PUTDEF establishes a command list (e.g., -COMS expression) and notices the file. For example,

```
←(PUTDEF 'TEST
         'FILES
         (LIST (LIST 'FNS '* 'TESTFNS)))
TEST
←FILELST
(TEST)
←TESTCOMS
((FNS * TESTFNS))
```

ADVISE
If TYPE is ADVISE, PUTDEF will put DEFINITION under the property READVICE on NAME's property list and invoke READVISE to advise the item (see Section 21.3).

17.5.3 Copying Definitions

You may copy a definition from one item to another via **COPYDEF**, which takes the format

```
Function:       COPYDEF

# Arguments:    5

Arguments:      1) the source atom, OLD
                2) the destination atom, NEW
                3) a type identifier, TYPE
                4) the location of the definition, SOURCE
                5) a list of options, OPTIONS

Value:          The name of the new object.
```

COPYDEF copies the appropriate definition from OLD to NEW. COPYDEF is usually used to produce a new copy of a file. When TYPE is FILES, COPYDEF performs the following operations:

1. Establishes the command list for the file ⟨NEW⟩COMS.
2. Notices NEW.
3. Modifies the individual -COMS expressions such as VARS and FNS (e.g., ⟨OLD⟩VARS becomes ⟨NEW⟩VARS).
4. Invokes MAKEFILE to create the new file.

Consider the following example:

```
←(COPYDEF 'TEST 'GOOD 'FILES)
GOOD

←FILELST
(GOOD TEST)

←GOODCOMS
((FNS * GOODFNS))
```

Applying COPYDEF to a function definition, we obtain:

```
←(COPYDEF 'SIGN 'LSEQGTO 'FNS 'COMPLEX)
LSEQGTO

←(PP LSEQGTO)
(LSEQGTO
     (LAMBDA (X)
          (COND
               ((GREATERP x 0.0) 1)
               ((LESSP x 0.0)   -1)
               (T 0))))
(LSEQGTO)
```

17.5.4 Deleting a Definition

You may delete the definition associated with an atom via **DELDEF**, which takes the format

```
Function:       DELDEF
# Arguments:    2
Arguments:      1) an item, NAME
                2) a type identifier, TYPE
Value:          NAME, if successful.
```

DELDEF removes the definition of TYPE associated with NAME that is currently in effect. Consider the following example:

```
←(DELDEF 'LSEQGTO 'FNS)
LSEQGTO

←(PP LSEQGTO)
(LSEQGTO not printable)
(LSEQGTO)
```

because the contents of the function definition cell have been erased.

```
←(SETQ x (LIST 'reagan 'bush 'mondale 'ferraro))
(reagan bush mondale ferraro)

←(DELDEF 'x 'VARS)
X

←x
UNBOUND ATOM
X
```

17.5.5 Showing Definitions

You may want to see how a definition would appear when it is written to a file. **SHOWDEF** prettyprints the definition for you. It takes the format

```
Function:       SHOWDEF
# Arguments:    3
Arguments:      1) an item, NAME
                2) a type identifier, TYPE
                3) a file name, FILE
Value:          The prettyprinted form of the definition.
```

SHOWDEF is used by several File Package functions when they do not know where to place a function. If you do not tell them, these functions show you the definition and re-ask you where to place it. It is primarly used by ADDTOFILES? (see Section 17.3.11).

Consider the following example:

```
←(SHOWDEF 'CZERO 'FNS)
(DEFINEQ
     (CZERO
          (LAMBDA NIL
               (COMPLEX 0.0 0.0))))
T
```

17.5.6 Editing a Definition

EDITDEF allows you to edit the definition of an item of a given type. It takes the form

```
Function:       EDITDEF

# Arguments:    4

Arguments:      1) an item, NAME
                2) a type identifier, TYPE
                3) the location of the definition, SOURCE
                4) an optional list of editor commands,
                   COMMANDS

Value:          The name of the object.
```

EDITDEF is equivalent to:

Using GETDEF to obtain the definition form SOURCE.
Applying the editing commands via EDITE to the definition.
Using PUTDEF to replace the definition at SOURCE.

EDITDEF is particularly convenient because you do not have to load the file contents (if the definition resides on an noticed file) in order to edit the definition. Moreover, no memory is consumed because the function is only loaded within the editor environment.

If SOURCE is NIL, EDITDEF uses the definition of the object in memory.

```
←(EDITDEF 'SLOTRECORD 'RECORD NIL '(R ID IDENTIFIER))
SLOTRECORD

←(RECLOOK 'SLOTRECORD)
(RECORD SLOTRECORD
        (IDENTIFIER VALUE TYPE RELATION OFFSPRING))
```

If COMMANDS is NIL, EDITDEF places you in the Editor where it waits to receive commands that you type in. See Chapter 19 for a discussion of the Editor.

17.5.7 Saving and Unsaving Definitions

You may save the definition of an item for later retrieval. You may either provide a definition as an argument or use the definition currently in effect. **SAVEDEF** takes the form

```
Function:       SAVEDEF
# Arguments:    3
Arguments:      1) an item, NAME
                2) a type identifier, TYPE
                3) a definition, DEFINITION
Value:          The name of the property under which the
                definition is saved.
```

SAVEDEF takes either DEF or the current definition in effect, if DEFINITION is NIL, and stores it such that it may later be retrieved by either GETDEF or UNSAVEDEF.

If TYPE is FNS or NIL (whence FNS is assumed), SAVEDEF stores the definition on the item's property list under the property EXPR, CODE, or SUBR depending on whether source or compiled format, e.g.,

```
←SAVEDEF 'SIGN 'FNS)
EXPR

←(GETPROPLIST 'SIGN)
(EXPR (LAMBDA (X) (COND ((GREATERP X 0.0) 1) ((LESSP X
0.0) -1) (T 0))))
```

If TYPE is VARS, the definition is stored on the property list under the property VALUE. Otherwise, it is saved in an internal data structure, SAVEDDEFS, e.g.,

```
←(SAVEDEF 'COMPLEX 'RECORD)
RECORDS

←SAVEDDEFS
((RECORDS (COMPLEX DATATYPE COMPLEX ((REAL FLOATP) (IMAG
FLOATP)))))
```

UNSAVEDEF is the corresponding function that allows you to unsave definitions that you have saved in the past. When you unsave a definition, the defini-

tion found on the property list or the internal data structure becomes the definition currently in effect for the item (i.e., it replaces the current definition in effect). UNSAVEDEF takes the form

```
Function:       UNSAVEDEF

# Arguments:    2

Arguments:      1) an item, NAME
                2) a type identifier, TYPE

Value:          The type identifier, if successful.
```

If TYPE is NIL, UNSAVEDEF looks for a definition under the EXPR property on the item's property list. If no EXPR property exists, it will look for a CODE or SUBR property, and use their values, respectively, as the definition. UNSAVEDEF recognizes EXPR, CODE, and SUBR as values for TYPE to indicate that it should explicitly unsave a value only for that property.

For example, we have saved the definition of SIGN under the EXPR property. Now, if we try to unsave a SUBR definition::

```
←(UNSAVEDEF 'SIGN 'SUBR)
(SUBR not found)
```

SAVEDEF and UNSAVEDEF are used by several INTERLISP subsystems to preserve the original body of the function. You may want to use them to preserve a definition via SAVEDEF before you modify it. Thus, if the modification is unsuccessful, you may recover the original definition via UNSAVEDEF.

17.5.8 Loading a Definition

You may load an arbitrary definition from the file using **LOADDEF**, which takes the form

```
Function:       LOADDEF

# Arguments:    3

Arguments:      1) the name of an object, NAME
                2) the type of the object, TYPE
                3) the source file, SOURCE

Value:          The definition of the object.
```

LOADDEF retrieves the definition of the object NAME with the given type from SOURCE and assigns that definition to the object in memory.

Consider the following example:

```
←(OPENFILE 'AMIS 'INPUT 'OLD)
[DSK]AMIS:5

←(LOADDEF 'SLOTRECORD 'RECORD 'AMIS)
SLOTRECORD

←(RECLOOK 'SLOTRECORD)
(RECORD SLOTRECORD
        (ID VALUE TYPE RELATION OFFSPRING))
```

A Definition for LOADDEF
We might define LOADDEF as follows:

```
(DEFINEQ
     (loaddef (name type source)
          (PUTDEF name
                  type
                  (GETDEF name
                          type
                          source
                          '(NODWIM NOCOPY)))
     ))
```

17.5.9 Renaming an Object

You may rename an object while preserving its value using **RENAME**, which takes the form

```
Function:       RENAME

# Arguments:    5

Arguments:      1) the old name, OLD
                2) the new name for the object, NEW
                3) a list of types for which objects are
                   to be renamed, TYPES
                4) a list of files where renaming will be
                   performed, FILES
                5) how the object will be renamed, METHOD

Value:          NEW, if successful.
```

RENAME performs its function in three steps:

1. Copies all instances of OLD to NEW for each type in TYPES.
2. Calls CHANGECALLERS to change all occurrences of OLD to NEW.
3. Deletes all instances of OLD via DELDEF.

Consider the following example. In our frame-based example, we have a function that we called CREATE.NODE. Suppose we want to change that function to CREATE.FRAME. We do so via the following function call:

```
←(LOAD 'FRAME)
⟨KAISLER⟩FRAME..1

←(RENAME 'CREATE.NODE 'CREATE.FRAME 'FNS)
⟨KAISLER⟩FRAME..1: 276 CREATE.NODE 8091
editing the variables SHKFNS:
CREATE.FRAME
```

FILES and METHOD are described in the following section.

17.5.10 Changing Calling Function Names

When editing a large application, you may want to change the name of a function to reflect a new capability, to reduce conflict with another name, or to be more semantic. However, the function may be utilized by many other functions in the program, particularly if it is a kernel function. Editing all functions that call the function can be a labor-intensive task. However, using Masterscope (see Chapter 26), INTERLISP can keep track of how a function is used and where it is used. **CHANGECALLERS** makes use of this knowledge about the structure of your program to perform these editing chores. It takes the form

```
Function:       CHANGECALLERS

# Arguments:    5

Arguments:      1) an old name, OLD
                2) a new name, NEW
                3) a list of object types, TYPES
                4) a list of files in which the objects
                   may occur, FILES
                5) an editing method, METHOD

Value:          NIL.
```

CHANGECALLERS finds all places where OLD is used as a type in TYPES. It changes all occurences of OLD to NEW in those locations. It also changes occurrences of OLD to NEW in all File Package commands so that you can remake the file easily. For example,

```
←(CHANGECALLERS 'ADD.TO.NET
                'ADD.TO.FRAME.LIST
                'FNS
```

```
                    NIL
                    'MASTERSCOPE)
.editing the functions CREATE.FRAME:
.editing the variables FRAMEFNS
.NIL
```

CHANGECALLERS determines if OLD might be used as more than one type in a program. If so, it asks you if you want to edit each occurrence type rather than automatically making the change for you. For example, in our complex arithmetic example, suppose I wanted to change the object COMPLEX to CMPLX. I would do so as follows:

```
←(CHANGECALLERS 'COMPLEX 'CMPLX)
Warning -- COMPLEX is also used as (RECORDS FILES)
⟨KAISLER⟩COMPLEX..2 263 299 362 COMPLEX CMULT CDIFFERENCE
CZERO CPLUS 3852 3926 3983 4070
editing the functions CMULT:
(COMPLEX (DIFFERENCE & &) (PLUS & &))
     Replace?
...
editing the records COMPLEX:
(DATATYPE COMPLEX (& &))
     Replace?
COMPLEX changed to CMPLX on COMPLEX
NIL
```

Note that you must also rename any object via RENAME if you use CHANGECALLERS as it only modifies those objects where the given object is used, but not the object itself.

METHOD provides you with two alternatives in making the changes:

1. If METHOD is EDITCALLERS, EDITCALLERS (see Section 19.1) is used to search FILES for all occurrences of OLD.
2. If METHOD is MASTERSCOPE (see Chapter 26), the Masterscope database, if any, is used to locate all occurrences of OLD. If no MASTERSCOPE database exists, it defaults to EDITCALLERS.

If METHOD is NIL, the default is MASTERSCOPE. However, the default may be specified by the system variable DEFAULTRENAMEMETHOD which is initially NIL. You may force a default to MASTERSCOPE by setting the value of DEFAULTRENAMEMETHOD to MASTERSCOPE and creating a Masterscope database. Otherwise, it defaults to EDITCALLERS.

17.5.11 Comparing Definitions

You may compare the definitions of two objects of a given type in two versions of a file, or you may compare all the definitions of a given name for all object types across all sources. Two functions support this capability: COMPARE and COMPAREDEFS.

COMPARE takes the form

```
Function:       COMPARE

# Arguments:    5

Arguments:      1) an object name, NAME-1
                2) an object name, NAME-2
                3) an object type, TYPE
                4) a source, SOURCE-1
                5) a source, SOURCE-2

Value:          T, if any differences; otherwise, NIL.
```

COMPARE prints a list of differences, if any, to your terminal. COMPARE invokes COMPARELISTS to perform a comparison of the two definitions. It might be defined as:

```
(DEFINEQ
   (compare (name-1 name-2 type source-1 source-2)
      (PROG (definition-1 definition-2)
            (SETQ definition-1
                  (GETDEF name-1 type source-1))
            (SETQ definition-2
                  (GETDEF name-2 type source-2))
            (COND
               ((COMPARELISTS definition-1 definition-2)
                  (*
                        If no differences between the
                        two definitions, merely return
                        NIL.
                  )
                  (RETURN)))
            (PRIN2 name-1 T T)
            (COND
               (source-1
                  (SPACES 1 T)
                  (PRIN1 "from" T)
                  (PRIN2 source-2 T T)))
            (SPACES 1 T)
```

```
            (PRIN1 "and" T)
            (PRIN2 name-2 T T)
            (COND
               (source-2
                    (SPACES 1 T)
                    (PRIN1 "from" T)
                    (PRIN2 source-2 T T)))
            (SPACES 1 T)
            (PRIN1 "differ" T)
            (TERPRI T)
            (COMPARELISTS definition-1 definition-2)
            (RETURN T))
    ))
```

COMPAREDEFS takes the form

```
Function:       COMPAREDEFS

# Arguments:    3

Arguments:      1) an object name, NAME
                2) a list of object types, TYPES
                3) a list of object sources, SOURCES

Value:          An indication of the comparison of all
                occurrences of NAME.
```

COMPAREDEFS uses COMPARELISTS to compare all occurrences of NAME as a type of TYPES in any source of SOURCES. It returns one of several values:

1. If NAME is not found in any source in SOURCES, it returns (QUOTE (NOT FOUND)).
2. If NAME occurs only once in any source of SOURCES, it returns (QUOTE (ONCE ONLY)), and there is no comparison to be made.
3. If COMPARELISTS detects any differences, which are printed to your terminal, it returns (QUOTE DIFFERENT).
4. Otherwise, it returns (QUOTE SAME).

17.5.12 Determining Type Existence

HASDEF allows you to determine if an object is a certain type. It takes the form

```
Function:       HASDEF

# Arguments:    4
```

```
Arguments:      1) an object, NAME
                2) a type specifier, TYPE
                3) an object source, SOURCE
                4) a spelling flag, SPELLFLAG

Value:          T, if NAME has a definition of TYPE.
```

If HASDEF does not find any object with NAME of TYPE, it attempts spelling correction on NAME, and returns the spelling-corrected name. Otherwise, it returns NIL. Spelling correction is attempted only if SPELLFLAG is T. Consider the example:

```
←(HASDEF 'CMULT 'FNS 'COMPLEX)
T

←(HASDEF 'CDIFERENCE 'FNS 'COMPLEX T)
=CDIFFERENCE
NIL
```

17.5.13 Determining the Types of an Object

TYPESOF allows you to determine the types of an object. It takes the form

```
Function:       TYPESOF

# Arguments:    4

Arguments:      1) an object, NAME
                2) a list of possible types, POSSIBLE-
                   TYPES
                3) a list of impossible types, IMPOSSIBLE-
                   TYPES
                4) an object source, SOURCE

Value:          A list of the types that are defined for
                NAME.
```

TYPESOF returns a list of the types in POSSIBLE-TYPES, but not in IMPOSSIBLE-TYPES, for which NAME has a definition. If POSSIBLE-TYPES is NIL, FILEPKGTYPES is used. Consider the example:

```
←(TYPESOF 'COMPLEX NIL NIL 'COMPLEX)
(COMPLEX -- no entry on USERMACROS)
(I.S.OPR COMPLEX not defined)
(no READVICE property for COMPLEX)
(RECORDS I.S.OPRS FNS FILES EXPRESSIONS ADVICE
FILEPKGCOMS)
```

Note that what TYPESOF returns is what COMPLEX could possibly be used as in COMPLEX, not how it is actually used.

A Definition for TYPESOF
We might define TYPESOF as follows:

```
(DEFINEQ
    (typesof (name)
        (for TYPE inside FILEPKGTYPES
            when
                (HASDEF name TYPE)
            collect TYPE)
    ))
```

17.6 DEFINING NEW FILE PACKAGE COMMANDS

The File Package includes a large number of types that have been developed and defined based on the experience of the INTERLISP developers. In most cases, these types will cover the data structures that you are likely to build in your programs. However, should you create unusual data structures, you may wish to build your own File Package commands to save and restore these data structures on a symbolic file.

17.6.1 FILEPKGCOM

FILEPKGCOM is the function that allows you to specify the characteristics of a new file package command. It takes the following form

```
Function:       FILEPKGCOM

# Arguments:    1-N

Arguments:      1) a command name, COMMAND
                2) a property name, PROPERTY
                3) a value for the property, VALUE
                4-N) property-value pairs

Value:          A list of the commands.
```

FILEPKGCOM is a Lambda, Nospread function that will define new file package commands or change the attributes of existing commands. The properties of the command are drawn from the following list:

```
MACRO           This property defines how to dump the
                command itself to a symbolic file. It
```

isused by MAKEFILE when writing an expression to the symbolic file. VALUE has the form

(⟨arguments⟩ . ⟨commands⟩)

The arguments given to the command name are substituted for the ⟨arguments⟩ throughout ⟨commands⟩. The result is a list of File Package commands.

For example,

```
(FILEPKGCOMS 'LISPXMACROS
                    'MACRO
                    '(X (COMS * (MAKELISPXMACROSCOMS .
                    X)))
                    'CONTENTS
                    'NILL)
```

ADD

This property specifies how to add an instance of an object to a given File Package command. It is used by ADDTOFILE to update the ⟨file⟩COMS list of a file. VALUE is a function, FN, of four arguments:

1. COM, a file package command
2. NAME, a typed object
3. TYPE, a type specifier
4. NEAR, a flag

FN should return T if it adds NAME to COM; otherwise, NIL.

Two possibilities exist:

1. COM has the form

   (⟨com⟩ * ⟨filevar⟩)

   whence NAME is added to (CADDR COM) which should be a literal atom.

2. COM has the form

   (⟨com⟩ ⟨varlist⟩)

   whence NAME is added to the (CDR COM).
3. TYPE is the type of the named object.
4. NEAR, if non-NIL, specifies near which item ADDTOFILE should attempt to insert NAME (see ADDTOFILES?).

DELETE

This property specifies how to delete an instance of an object from a File Package command. It is used by DELFROMFILES (see Section 17.7.2). VALUE is FN which is a function of three arguments corresponding to the first three arguments of ADD. FN returns T if it deletes NAME from COM, else NIL. If it returns ALL, COM should be interpreted as empty and may be deleted entirely from the command list.

CONTENTS

This property specifies whether or not an instance of an object of a given type is contained within a given File Package command. It is used by WHEREIS (see Section 17.3.8) and INFILECOMS? (see Section 17.7.3). VALUE is FN which is a function of three arguments corresponding to the three arguments of ADD.

NAME is interpreted as follows:

1. If NAME is NIL, FN returns a list of elements of type TYPE contained in COM.
2. If NAME is T, FN returns T if there are any elements of type TYPE in COM.
3. If NAME is any literal atom other than T or NIL, FN returns T if

NAME of type TYPE is contained in COM.

4. If NAME is a list, FN returns a list of those elements of type TYPE contained in COM that are also contained in NAME.

17.7 MANIPULATING FILE PACKAGE COMMANDS

How a file is to be loaded is described by the File Package commands. A list of commands is usually stored as the value of the variable ⟨filename⟩COMS. To create a file, you establish the content of ⟨filename⟩COMS and execute MAKEFILE (see Section 17.3.1). It uses the expressions found in the ⟨filename⟩COMS expression as a guide to writing the symbolic form of the file.

File Package commands may be very complex for large files containing many functions and initializing many variables. Moreover, the creation of unusual structures often requires some ingenuity in describing them using standard data structures or writing new functions to properly write them out and read them in. Managing these command lists may be quite difficult. To ease the burden, the File Package provides a set of functions for manipulating the ⟨filename⟩COMS lists.

Note that most if these functions are undoable.

17.7.1 Adding To a File's COMS

ADDTOFILE adds an entry to the COMS list of a file. It takes the form

Function:	ADDTOFILE
# Arguments:	3
Arguments:	1) an object, NAME 2) a file name, FILE 3) a command type, TYPE
Value:	The file name.

ADDTOFILE adds NAME of TYPE to the file package commands for FILE. For example, suppose I had an ugly variable that I wished to add to the file SHK.

```
←(ADDTOFILE 'x 'shk 'uglyvars)
SHK

←SHKCOMS
((FNS * SHKFNS) (VARS * SHKVARS) (UGLYVARS X))
```

Adding to a Command List

ADDTOCOMS adds an entry to a COMS list. It takes the form

```
Function:       ADDTOCOMS
# Arguments:    3
Arguments:      1) a command list, COMS
                2) an object, NAME
                3) a type, TYPE
Value:          T.
```

The command list, COMS, may take one of two values:

1. A variable whose value is a list of file package commands, i.e., MAPCOMS.
2. A list of file package commands.

```
←(ADDTOCOMS SHKCOMS 'X 'RECORD)
NIL
```

because ADDTOCOMS was unable to find a command of type RECORD in which to add X. On the other hand,

```
←(ADDTOCOMS SHKCOMS 'Y 'VARS)
T
←SHKCOMS
((FNS * SHKFNS) (VARS * SHKVARS) (UGLYVARS X))
←SHKVARS
(CURRENT.NETWORK.NODE Y)
```

ADDTOCOMS "understands" how to add an entry to a particular file package command based on its form.

17.7.2 Deleting from a File's COMS

DELFROMFILES deletes all instances of an object of a given type from the File Package commands associated with a file name. It takes the form

```
Function:       DELFROMFILES
# Arguments:    3
Arguments:      1) the name of an object, NAME
                2) an object type, TYPE
                3) a list of files, FILES
```

```
Value:          A list of files from which NAME has been
                deleted.
```

DELFROMFILES inspects the file package commands of all files that are named in FILES. It deletes NAME of TYPE from the appropriate commands in which it appears. If FILES is NIL, DELFROMFILES uses FILELST as its default value. FILES may also be an atom whence (LIST FILES) is used.

Consider the following example:

```
←(DELFROMFILES 'X 'UGLYVARS (LIST 'SHK))
(SHK)

←SHKCOMS
((FNS * SHKFNS) (VARS * SHKVARS)(UGLYVARS))
```

Deleting from a Command List

DELFROMCOMS is used by DELFROMFILES to delete an object from a specific file package command list. It takes the form

```
Function:       DELFROMCOMS

# Arguments:    3

Arguments:      1) a command list, COMS
                2) an object name, NAME
                3) an object type, TYPE

Value:          T.
```

COMS is usually the name of a ⟨file⟩COMS variable that contains the definition of the File Package commands for a file. DELFROMCOMS removes NAME from the specific command (e.g., sublist) within COMS according to its type.

Consider the following example:

```
←(DELFROMCOMS SHKCOMS 'Y 'VARS)
T

←SHKVARS
(CURRENT.NETWORK.NODE)
```

17.7.3 Determining if an Object is in a Command

INFILECOMS? searches a File Package command list to determine if an object of a given type appears in the command list. It takes the form

```
Function:       INFILECOMS?

# Arguments:    3
```

```
Arguments:      1) an object name, NAME
                2) a command list, COMS
                3) an object type, TYPE
Value:          T, if NAME of TYPE appears in COMS.
```

COMS is usually a variable whose value is a list of File Package commands. INFILECOMS? searches its value looking for an instance of NAME in a command associated with TYPE. It returns T if NAME is found. Consider the example:

```
←(INFILECOMS? 'MEDIAN AMISUTILCOMS 'FNS)
T
```

If NAME has the value NIL, INFILECOMS? will return a list of all objects with the type TYPE. Consider the example:

```
←(INFILECOMS? NIL AMISUTILCOMS 'FNS)
(INSERT.STRING DELETE.STRING SUBSTITUTE.STRING
MAKE.STRING.FROM.LIST ...)
```

If NAME has the value T, INFILECOMS? returns T if there are any objects of type TYPE in the command list.

```
←(INFILECOMS? T AMISUTILCOMS 'RECORD)
NIL
```

Obtaining A List of Object Types

FILECOMSLST returns a list of the objects of type TYPE that appear in a file. It takes the form

```
Function:       FILECOMSLST
# Arguments:    2
Arguments:      1) a file name, FILE
                2) an object type, TYPE
Value:          A list of objects of type TYPE.
```

FILECOMSLST searches the File Package commands associated with FILE for objects of type TYPE. It returns a list of all such objects, e.g.,

```
←(FILECOMSLST 'AMIS 'RECORDS)
(GRAPH.NODE SLOTRECORD DEMONRECORD)
```

Obtaining a List of Functions in a File

We frequently ask for a list of all the functions in a file. **FILEFNSLST** returns a list of all functions that are named in File Package commands. It takes the form

```
Function:       FILEFNSLST
# Arguments:    1
Argument:       1) a file name, FILE
Value:          A list of functions named in the File
                Package commands.
```

Consider the following example:

```
←(FILEFNSLST 'AMIS)
(CREATE.DATABASE.FUNCTIONS.MENU CREATE.DATA.MODELS
MAKE.DB.COMS MAKE.DB.FILENAME ...)
```

17.7.4 Making a New File Package Command

MAKENEWCOM creates a new file package command that will dump an object of a type to a file when the file is (re)made. It takes the form

```
Function:       MAKENEWCOM
# Arguments:    2
Arguments:      1) an object name, NAME
                2) an object type, TYPE
Value:          The File Package command for dumping NAME
                of TYPE to a file.
```

The command returned by MAKENEWCOM is usually appended to the File Package commands associated with a file, e.g.,

```
←(SETQ AMISUTILCOMS
       (APPEND AMISUTILCOMS (MAKENEWCOM 'VARIANCE
'FNS)))
(.....
     (FNS VARIANCE))
```

17.7.5 Creating a COMS Variable Name

An operation that must frequently be performed is to create the name of the variable whose value is the list of File Package commands. This operation is

required because many File Package functions require that you only submit the name of the file. **FILECOMS** takes the form

```
Function:       FILECOMS
# Arguments:    2
Arguments:      1) a file name, FILE
                2) an object type, TYPE
Value:          An atom which is a variable name.
```

If TYPE is NIL, then FILECOMS returns an atom of the form ⟨file⟩COMS. Otherwise, it returns an atom composed from the file name and the object type. Consider the examples:

```
←(FILECOMS 'AMISMODELS)
AMISMODELSCOMS
←(FILECOMS 'AMISNET 'VARS)
AMISNETVARS
```

17.7.6 Smashing a File's COMS

SMASHFILECOMS sets all file variables to NOBIND in a command list. It takes the form

```
Function:       SMASHFILECOMS
# Arguments:    1
Argument:       1) a file name, FILE
Value:          The file name.
```

SMASHFILECOMS sets all file variables that appear in File Package commands to NOBIND. These variables take the form

```
(⟨command⟩ * ⟨file variable⟩)
```

It also sets the name of the command list variable to NOBIND. Consider the following example:

```
←SHKCOMS
((FNS * SHKFNS) (VARS * SHKVARS))
←SHKFNS
(CREATE.FRAME INITIALIZE.FRAME DEFINE.ATTRIBUTE ...)
```

```
←SHKVARS
(CURRENT.NETWORK.NODE)

←(SMASHFILECOMS 'SHK)
SHK

←SHKCOMS
UNBOUND ATOM
SHKCOMS
```

as are SHKFNS and SHKVARS.

17.7.7 Moving an Item Between Files

MOVETOFILE allows you to move an object of a given type from one file to another. It takes the form

```
Function:       MOVETOFILE

# Arguments:    4

Arguments:      1) a destination file, DSTFILE
                2) an object, NAME
                3) an object type, TYPE
                4) a source file, SRCFILE

Value:          The object name.
```

MOVETOFILE is a shorthand notation for using DELFROMFILE and ADDTOFILE in the proper way to update the File Package commands of the two files.

17.8 PRETTYPRINTING SYMBOLIC FILES

In Section 15.7, we examined the functions for prettyprinting function definitions to a file or a terminal. When you make a symbolic file, the S-expressions comprising the function definitions and other forms are compressed to eliminate superfluous blanks and end-of-line characters. While storage efficient, these files are extremely difficult to read. You may create a symbolic file in prettyprinted format using the **PRETTYDEF** package.

17.8.1 Prettyprinting Function Definitions

PRETTYDEF writes a symbolic file, suitable for loading, in prettyprint format. It has the following form

```
Function:       PRETTYDEF

# Arguments:    6
```

```
Arguments:      1) an list of functions, PRETTYFNS
                2) a file name, PRETTYFILE
                3) file package commands, PRETTYCOMS
                4) a list of functions, REPRINTFNS
                5) a file name, SOURCEFILE
                6) a list, CHANGES

Value:          The full name of the symbolic file.
```

PRETTYDEF creates a symbolic file using the value of PRETTYFILE. If PRETTYFILE is NIL, the primary output file is used. If PRETTYFILE is an atom, its value is used as the file name. If the symbolic file is not already open, it will be opened by PRETTYDEF. It is automatically closed when PRETTYDEF terminates.

```
←(PRETTYDEF (LIST 'IMAG))
(DEFINEQ
     (IMAG (LAMBDA (CX) **COMMENT**
           (RECORDACCESS (QUOTE IMAG)
                         CX NIL (QUOTE FETCH)))))
T
```

where the primary output file is used.

If PRETTYFILE is a list, the symbolic file name is assumed to be the CAR of the list. Processing is the same except that the file will not be closed when PRETTYDEF terminates.

PRETTYFNS is an optional list of functions that are to be printed on the symbolic file. If PRTTYFNS is NIL, PRETTYDEF assumes the form (FNS * ⟨prttyfns⟩) will appear in the file package commands.

```
←(PRETTYDEF NIL NIL COMPLEXCOMS NIL 'COMPLEX)
(RPAQQ COMPLEXCOMS (COMPLEX REAL IMAG CPLUS CDIFFERENCE
...
                   RECIPROCAL TRUNCATE PRINT.ARRAY))
(DEFINEQ
     (COMPLEX (LAMBDA (R I) **COMMENT**
...
```

All of the functions in COMPLEXCOMS will be prettyprinted to the primary output file, since no functions are specifically identified.

PRETTYCOMS is a list of file package commands. If PRETTYCOMS is an atom, its value is used. PRETTYDEF writes an RPAQQ on the file which sets the atom to the command list when the file is loaded.

REPRINTFNS and SOURCEFILE are used when a file is remade (see Section 17.3.2). REPRINTFNS is a list of functions to be printed. If its value is

EXPRS, all functions with EXPR definitions will be prettyprinted. If its value is ALL, all functions with EXPR definitions or EXPR properties will be printed.

SOURCEFILE is the name of a file from which to copy the definitions of those functions that are not going to be prettyprinted, i.e., those not specified by REPRINTFNS. If SOURCEFILE is T, the most recent version of PRETTYFILE is used. Thus, when you make the next version of PRETTYFILE, you can directly copy the definitions of any functions that have not changed from the previous version. If there is no previous version, PRETTYDEF prints the message "⟨prettyfile⟩ NOT FOUND, SO IT WILL BE WRITTEN ANEW". It then proceeds as if REPRINTFNS and SOURCEFILE were both NIL.

CHANGES is for internal use by the File Package.

A Definition for PRETTYDEF

We might define a simple version of PRETTYDEF as follows:

```
(DEFINEQ
    (prettydef (prettyfns prettyfile prettycoms)
        (PROG (save-output-file closefile?
        prettycomlist prettydate)
              (*
                  Save the output file name.
              *)
              (SETQ save-output-file (OUTPUT))
              (COND
                  ((LISTP prettyfile)
                       (*
                            If PRETTYFILE is a list,
                            get its CAR which is the
                            actual file name, and note
                            that it should not be
                            closed when PRETTYDEF
                            terminates.
                       *)
                       (SETQ prettyfile (CAR
                       prettyfile))
                       (SETQ closefile? T)))
              (*
                  If PRETTYFNS is NIL or an atom which
                  is not a list and PRETTYCOMS is NIL
                  or an atom which is not a list, there
                  is no specification from which
                  PRETTYDEF can work. Declare an error.
              )
              (AND
                  (NLISTP prettyfns)
```

```
        (NLISTP (EVALV prettyfns))
        (NLISTP prettycoms)
        (NLISTP (EVALV prettycoms))
        (ERROR "ILLEGAL OR MEANINGLESS CALL:"
               (LIST 'PRETTYDEF
                     prettyfns
                     prettyfile
                     prettycoms)))
(COND
    ((NULL prettyfile)
         (*
              If PRETTYFILE is NIL,
              output is directed to the
              primary output file, which
              you cannot close.
         )
         (SETQ closefile? T))
    ((OPENP prettyfile 'OUTPUT)
         (*
              Attempt to open the output
              file.
         )
         (OUTPUT prettyfile))
    (T
         (OUTFILE prettyfile)
         (SETQ prettydate
                (PRINTDATE
                   (NAMEFIELD
                   prettyfile)
                   'PRETTYDEF))))
(SETQ prettyfile (OUTPUT))
(COND
    ((OR
         (LISTP prettyfns)
         (LISTP (CAR prettyfns)))
              (PRINTFNS prettyfns)
              (PRETTYCOM prettyfns T)))
(COND
    ((AND
         (NLISTP prettycoms)
         (NLISTP (CAR prettycoms)))
         (GO out)))
(*
    Print the File Package commands in a
    prettyprinted format. Each command
```

```
                type must be treated according to its
                own characteristics.
            )
            (PRETTYCOM prettycoms T)
            (COND
              ((ATOM prettycoms)
                (SETQ prettycoms (CAR prettycoms))))
            (SETQ prettycomlist prettycoms)
    loop
            (COND
                ((NLISTP prettycomlist)
                    (GO out)))
            (PRETTYCOM (CAR prettycomlist))
            (SETQ prettyl (CDR prettycomlist))
            (GO loop)
    out
            (*
                Restore the primary output file.
            )
            (OUTPUT saveout)
            (COND
                ((NULL closefile?)
                    (*
                        If NIL, close the output
                        file.
                    )
                    (ENDFILE prettyfile)
                    (EVAL prettydate)))
            (RETURN prettyfile))
))
```

17.8.2 Printing a Definition

PRINTDEF prints any expression in pretty format on the primary output file. It takes the form

```
Function:       PRINTDEF

# Arguments:    6

Arguments:      1) an expression, EXPRESSION
                2) a left hand margin, LEFTMARGIN
                3) a definition flag, DEFFLG
                4) a tail flag, TAILFLG
```

```
                5) a font list, FONTLST
                6) a file name, FILE

Value:          NIL.
```

PRINTDEF usually prints the value of EXPRESSION on the primary output file using the primary readtable. If FILE is non-NIL, the output from PRINTDEF will be directed to the specified file, which must be opened for output.

LEFTMARGIN determines the number of spaces from the logical left end of the output display. Thus,

```
←(PRINTDEF ⟨expression⟩)
```

will result in the expression being printed abutting the left edge of your display terminal. The value of (LINELENGTH) determines the right hand margin.

If DEFFLG has the value T, then the value of EXPRESSION is treated as a function definition. Special action is taken for LAMBDAs, PROGs, CONDs, comments, and CLISP words. Otherwise, no special action is taken. DEFFLAG is NIL when PRINTDEF calls PRETTYPRINT to print variables and property lists or when the editor calls PRINTDEF via PPV.

If TAILFLG has the value T, then EXPRESSION is treated as the tail of a list which should be printed without parentheses.

FONTLST is used by the Font Package which is not discussed in this text.

Consider the following example:

```
←(PRINTDEF complexcoms 10 NIL T)
         (FNS * COMPLEXFNS)
         (RECORDS COMPLEX)
         (P (DEFPRINT (QUOTE COMPLEX)
                      (FUNCTION PRINT.COMPLEX)))
NIL
←(PRINTDEF (GETD 'RECIPROCAL) 5 T T)
     LAMBDA (X) **COMMENT**
     (COND
        ((EQP X 0.0)
              (ERROR "ZERO HAS NO RECIPROCAL"))
        (T
              (QUOTIENT 1.0 (FLOAT X))))
NIL
```

17.8.3 Making a File Creation Slug

FILECREATED prints a message followed by the time and date the file was made. FILECREATED takes the following form

```
Function:       FILECREATED
# Arguments:    1
Argument:       1) an expression, EXPRESSION
Value:          NIL.
```

FILECREATED is an NLAMBDA, nospread function.

The time and date of file creation are found in the CAR of EXPRESSION. The message is the value of the variable PRETTYHEADER, which is initially set to the string "FILE CREATED:". If PRETTYHEADER is NIL, nothing will be printed.

The CDR of EXPRESSION contains information about the file. Since FILECREATED is usually called by PRINTDATE, this information is already properly formatted.

FILECREATED stores the time and date the file was made on the file's property list under the property FILEDATE.

Consider the following example:

```
←(SETQ file-info (CONS (DATE) NIL)))
("18-SEP-84 20:09:06")
←(APPLY (FUNCTION FILECREATED) file-info)
FILE CREATED 18-SEP-84 20:09:06
NIL
```

17.8.4 Obtaining the File Date

You may obtain the file date by invoking **FILEDATE**, which takes the following form

```
Function:       FILEDATE
# Arguments:    1
Argument:       1) a file name, FILE
Value:          The file date in the FILE CREATED
                expression.
```

FILEDATE returns the date contained in the FILE CREATED expression stored in the file header, e.g.,

```
←(FILEDATE 'COMPLEX)
"30-AUG-84 20:44:36"
```

17.8.5 Printing the File Date

PRINTDATE is usually called by PRETTYDEF to insert the "FILE CREATED" expression at the beginning of a symbolic file. It takes the following format

```
Function:       PRINTDATE
# Arguments:    2
Argument:       1) a file name, FILE
                2) a list of changes, CHANGES
Value:          The expression which is the CONS of the
                date and CHANGES.
```

CHANGES is used by the File Package commands to insert an expression of the form

```
changes to: (FNS ...)
```

in the "FILE CREATED" expression.

PRINTDATE calls upon **DATE** (see Section 29.1.1) to return the current date and time as a string suitable for printing in a symbolic file.

Consider the following example:

```
←(PRINTDATE T NIL)
(FILECREATED "18-SEP-84 20:17:52" T)
(("18-SEP-84 20:17:52" T))
```

Note that if there are no changes to the file, **PRINTDATE** inserts the value T as a placeholder.

An alternative definition for **PRINTDATE** which provides more information might be defined by you as:

```
(DEFINEQ
(printdate (file format source-date previous-source-date
            compile-date previous-compile-date)
     (PROG (slug)
           (SETQ slug
                 (NCONC
                    (LIST 'FILEHEADER
                          file
                          (LIST 'DATE (DATE))
                          (LIST 'FORMAT
                                (OR format "SPECIAL"))
                          (LIST 'FILENAME
```

```
                    (FILEVERSION (OUTPUT)))
                (LIST 'SOURCE-DATE
                    (OR source-date (DATE))))
            (COND
              (previous-source-date
                (LIST
                  (LIST 'PREVIOUS-SOURCE-DATE
                      previous-source-date))))
            (AND compile-date
                (LIST
                    (LIST 'COMPILING-DATE
                        compile-date)))
            (AND previous-compile-date
                (LIST
                    (LIST 'PREVIOUS-COMPILE-
                    DATE
                        previous-compile-
                        date)))
            ))
        (PRINTDEF1 slug)
        (RETURN slug))
))
```

17.8.6 Printing Function Definitions on a File

PRINTFNS prints DEFINEQ on the primary output file. It then prettyprints the definitions of functions which are given as its argument. It takes the form

```
Function:       PRINTFNS
# Arguments:    1
Argument:       1) a list of functions, FNSLST
Value:          NIL.
```

Usually, PRINTFNS is given a variable of the form ⟨file⟩FNS from a File Package command which has the form:

```
(FNS * ⟨file⟩FNS)
```

Alternatively, if the function names are distributed in the File Package commands in the form:

```
(FNS ⟨filename1⟩ ... ⟨filenameN⟩)
```

PRINTFNS may be called with (CDR ⟨expression⟩). Yet a third form is

```
(PRINTFNS (FILEFNSLST ⟨filename⟩))
```

which gathers into a list all of the function names defined in the file.

Consider the following example:

```
←(PRINTFNS (LIST 'ROUNDTO))
(DEFINEQ
    (ROUNDTO
        (LAMBDA (X Y) **COMMENT**
            (TIMES (ROUNDED (QUOTIENT X Y)
                Y)))))
NIL
←(PRINTFNS 'ADD3)
(DEFINEQ)
NIL
```

17.8.7 Printing a COMS Message upon Loading

PRETTYCOMPRINT prints the value of its argument using LISPXPRINT unless the value of PRETTYHEADER is NIL. Usually, its argument is the name of the variable containing the File Package commands for a file. It takes the form

```
Function:       PRETTYCOMPRINT
# Arguments:    1
Argument:       1) an expression, EXPRESSION
Value:          NIL.
```

PRETTYCOMPRINT is an NLAMBDA function. It prints the value of EXPRESSION.

When you load a file, LOAD prints certain information about the file name and when it was created. It then begins executing S-expressions that it reads from the file. Normally, the first of these S-expressions displays the File Package command variable.

If PRETTYHEADER is NIL, nothing will be printed by PRETTYCOMPRINT. Thus, if you want to load a file without alerting the user that you have done so, you might temporarily bind PRETTYHEADER to NIL while you load the file.

17.8.8 Obtaining the File Changes

FILECHANGES prints the changes to the file stored in the file header expression at the beginning of the file. It takes the form

```
Function:       FILECHANGES

# Arguments:    2

Arguments:      1) a file name, FILE
                2) a file package type, TYPE

Value:          A list of the changed objects of type
                TYPE.
```

FILECHANGES inspects the changes expression at the beginning of the file. It returns a list of all the changes of the given type. If TYPE is NIL, FILECHANGES returns an association list (alist) of all changes where the file package type is the CAR of the alist.

```
←(FILECHANGES 'FRAME)
((FNS ADD.TO.FRAME.LIST CREATE.FRAME INITIALIZE.FRAME)
(VARS SHKFNS))
```

17.9 SYMBOLIC FILE INPUT

Programs are usually stored in source form in symbolic files on mass storage. To create an Interlisp application environment, we load one or more symbolic files into the Interlisp virtual memory. Symbolic files are created in two ways:

1. Using a text editor to type in source code.
2. Using MAKEFILE to write out function definitions and other information as directed by File Package commands.

As we have seen in previous sections, the File Package writes a considerable amount of information describing an application environment into a file. As applications grow in complexity, a symbolic file may contain a large number of functions. It may be time-consuming to load all of the information from the file. The File Package provides a number of functions for loading all or portions of a symbolic file.

17.9.1 Generalized Load

LOAD is the generalized loading function. It reads successive S-expressions from a file until it encounters the atom STOP. Each S-expression is evaluated as it is read. It takes the form

```
Function:       LOAD
                LOAD?

# Arguments:    3
```

```
Arguments:      1) a file name, FILE
                2) a load flag, LDFLG
                3) a print flag, PRINTFLAG
Value:          The file name or NIL.
```

LOAD expects the file to exist when it accesses it. If it does not, LOAD prints the message FILE NOT FOUND followed by the file name.

If LOAD encounters an end-of-file condition, it displays an error message ? and breaks.

Because each S-expression is evaluated as it is read, erroneous S-expressions can cause a break to occur. Depending on how you handle the break, loading may not be completed.

PRINTFLAG may be used to trace the loading of S-expressions. If PRINTFLAG has the value T, LOAD prints the value of each S-expression; otherwise, it does not. Consider the following example:

```
←(LOAD 'COMPLEX T T)
⟨KAISLER⟩COMPLEX..2
FILE CREATED 30-AUG-84 20:44:36
NIL
COMPLEXCOMS
COMPLEXCOMS
((FNS * COMPLEXFNS) (RECORDS COMPLEX) ...)
(COMPLEX REAL IMAG CPLUS CMULT ...)
...
⟨KAISLER⟩COMPLEX..2
```

LDFLG affects the evaluation of DEFINE(Q) and RPAQ(Q) functions. DFNFLG (see Section 8.2.4) is bound to LDFLG while LOAD executes. LDFLG may take the values:

NIL
If LDFLG is NIL, and a function is redefined (e.g., a definition already exists in the virtual memory), LOAD displays the message (⟨function-name⟩ REDEFINED) and saves the old definition on the atom's property list. Thus, you may recover the original (in memory!) definition of the function using UNSAVEDEF.

T
If LDFLG is T, the old definition is overwritten. You are not informed that the function has been overwritten.

PROP
If LDFLG has the value PROP, function definitions are stored on the atom's property list under the property EXPR. This action is the corollary to (1) where the old definition is saved. Here, the new definition is saved on the property list.

ALLPROP
If LDFLG has the value ALLPROP, both function definitions and variables set by RPAQ(Q) are stored on property lists.

SYSLOAD
LDFLG has the value SYSLOAD, LOAD does not load debugging and development information. The resulting application will execute more efficiently. LOAD takes the following actions:

a. Binds DFNFLG to T.
b. Binds LISPXHIST to NIL so that the loading operation is not undoable (see Section 28.2).
c. Binds ADDSPELLFLG to NIL to suppress adding names to spelling lists. (see Section 22.7.2).
d. Binds FILEPKGFLG to NIL so that the file is not noticed by the File Package.
e. Binds BUILDMAPFLG to NIL to prevent construction of a file map.
f. Binds the -COMS and -VARS entries to NOBIND.
g. Adds FILE to SYSFILES rather than FILELST.

An alternative form, **LOAD?**, operates the same except that it does not load the file if it has already been loaded. In this case, its value is NIL.

17.9.2 Loading Selected Functions

Rather than loading an entire file, you may load selected functions and variables from a file. **LOADFNS** takes the form

```
Function:       LOADFNS
# Arguments:    4
Arguments:      1) a list of functions, FNS
                2) a file name, FILE
                3) a load flag, LDFLG
                4) an S-expression, VARS
Value:          A list of functions found and loaded as
                well as those not found.
```

FNS may be a list of functions, or a single function name, or T. The latter case forces all functions to be loaded. FILE may be either a compiled or symbolic file. LDFLG is handled exactly the same as in LOAD.

VARS specifies how to handle expressions other than function definitions. There are several possible values:

1. T, which loads all expressions.
2. NIL, which load none of the expressions.
3. VARS, which loads only expressions created by RPAQ(Q).
4. FNS/VARS, which loads only -COMS and -BLOCKS expressions.
5. Any other atom is treated as (LIST ⟨atom⟩).
6. If VARS is a list, all atoms in the list are compared to the CARs of each expression. Wherever a match is found, the expression will be loaded.

LOADFNS returns a list of functions loaded as well as those not found. The format of the list is:

```
(⟨loaded-functions⟩ (NOT-FOUND: ⟨not-found-functions⟩))
```

If VARS was non-NIL, the list will also include those expressions that were loaded as well as those members of VARS (as a list) which were not found.

If FILE is NIL, WHEREIS (see Section 17.3.8) is invoked to locate the file containing the first function in FNS. It assumes that all functions will be loaded from that file.

17.9.3 Loading Selected Expressions

LOADVARS loads S-expressions from a file. It takes the form

```
Function:       LOADVARS
# Arguments:    3
Arguments:      1) a variable list, VARS
                2) a file name, FILE
                3) a load flag, LDFLG
Value:          A list of variables loaded.
```

This form is equivalent to

```
(LOADFNS NIL ⟨file⟩ ⟨ldflg⟩ ⟨vars⟩).
```

17.9.4 Editing Functions Without Loading

When the File Package has noticed a file, it knows about its contents. At that point, you can edit functions within a file without consuming memory to load them. To notice a file in this manner, you may use **LOADFROM**, which takes the form

```
Function:       LOADFROM
# Arguments:    3
```

```
Arguments:      1) a file name, FILE
                2) a list of functions, FNS
                3) a load flag, LDFLG

Value:          The file name.
```

Typically, LOADFROM is used to notice a file, e.g.,

```
←(LOADFROM 'ADV15.DCOM)
[KAISLER]ADV15.DCOM;2
```

You may load a few functions in addition to noticing the file by providing a second argument. LOADFROM is equivalent to:

```
(LOADFNS ⟨fns⟩ ⟨file⟩ ⟨ldflg⟩ T).
```

17.9.5 Loading from a Compiled File

LOADCOMP loads expressions from a compiled, rather than symbolic, file. It takes the form

```
Function:       LOADCOMP

# Arguments:    2

Arguments:      1) a file name, FILE
                2) a load flag, LDFLG

Value:          The file name.
```

LOADCOMP performs all the functions on a file associated with compilation. That is, it evaluates all expressions included in DECLARE: EVAL@COMPILE expressions. It notices the function and variable names by adding them to the lists NOFIXFNSLST and NOFIXVARSLST.

18

Error Handling

Errors occur frequently during programming, particularly in the early stages of program development. During this period, easy handling of errors is a necessity because it is likely that you will be encountering many of them. As a program ages, most of the obvious errors are eliminated. It is the subtle errors, mainly due to logic or complex data structures, that begin to plague the programmer. Now, what is needed is not easy handling, but powerful tools to diagnose the cause of the error and determine the corrective action. The process of isolating the error and finding the fix is often labor-intensive in these latter stages of program development.

INTERLISP provides a powerful facility for catching and handling errors as they occur. Unlike other programming languages where the user may merely inspect the state of the program after an error occurs, INTERLISP allows the user to interactively correct the problem and continue the computation. INTERLISP also provides a set of tools that can assist you in isolating errors within program segments and identifying their causes. This duality of capability is one of the reasons that INTERLISP is a powerful and flexible program development environment.

18.1 HOW ERRORS OCCUR

Errors can occur for many reasons. A discussion of error types is presented in Section 18.5. However, because of the simplicity of datatypes and representations in INTERLISP, errors frequently occur for a few well-defined reasons.

The most frequent case is an unbound atom. When atoms are created, their value cell is filled with the atom NOBIND to indicate that they have not been assigned a value. When the interpreter encounters an atom whose value cell contains NOBIND, it generates an error. It does so by calling the function FAULTEVAL. FAULTEVAL will print the character string "U.B.A." to indicate an unbound atom and then break.

The second most frequent cause of errors is an attempt to invoke a function which is undefined. This error is often due to a misspelling either in the function definition or, more likely, in the statement where the function is to be invoked. Another case occurs when the interpreter encounters a list in EVAL mode whose CAR is not the name of a function. The interpreter calls FAULTEVAL which normally prints "U.D.F.", to indicate an undefined function, and then breaks.

Other types of errors are usually specific instances of these generic errors. They are specifically identified because they usually occur in primitive functions and occur frequently in the early stages of program development. Specializing the error helps you identify the cause more rapidly. There are over fifty different errors currently detected in INTERLISP. Some are specific to the implementation while others are detected in every implementation. Section 18.5 discusses the generic errors. Most of these errors will cause a break to occur and place your computation under the control of the Break Package.

Once inside the Break Package, the user may perform any number of operations to diagnose and correct the error. These operations are discussed in Chapter 20.

18.2 CATCHING AND HANDLING ERRORS

A good dose of preventative medicine often prevents a program from going awry and causing significant damage to your environment or computation. Catching errors as they occur but before they have a deleterious effect allows you to prevent any damage that might occur. This approach is useful for two reasons:

1. It makes your programs user-friendly if you can alert the user when an error occurs, provide him with information on how to correct it, and allow him to do so.
2. When you attempt to account for errors in your program, it makes you think carefully about potential faults and plan for their occurrence. This is good software engineering practice.

INTERLISP provides a mechanism for catching a subset of the errors that can occur before they cause a break. When the error is caught, the user may analyze it and decide whether or not to break, break sometimes (depending on other factors), or take corrective action. Another mechanism allows a user to provisionally execute a function, determine whether an error would occur or not, and proceed based on this information. These two mechanisms are discussed in the following sections.

18.2.1 Catching Errors: ERRORTYPELST

ERRORTYPELST is a system variable that is used by FAULTEVAL to determine how an error should be handled. ERRORTYPELST is a list of lists each having the format

```
(⟨number⟩ ⟨expression1⟩ ... ⟨expressionN⟩)
```

The ⟨number⟩ is one of the error numbers described in Section 18.5. The ⟨expression⟩s are INTERLISP code for handling the error. If an error number does not appear on ERRORTYPELST, then an error of that type will always cause a break.

ERRORTYPELST is used by BREAKCHECK to determine whether or not an error should cause an invocation of FAULTEVAL. ERRORTYPELST is searched for an entry with a CAR corresponding to the type of error that has occurred. If it is found, then the ⟨expression⟩s are evaluated. If the last ⟨expression⟩ produces a non-NIL value, that value is substituted at the point where the error occurred and the function is re-entered.

Consider the following example [irm83] which defines ERRORTYPELST to have an entry for catching errors of the type NON-NUMERIC ARG. The form of the entry is

```
(SETQ ERRORTYPELST
      (10
          (AND
               (NULL (CADR ERRORMESS))
               (SELECTQ (STKNAME ERRORPOS)
                    ((IPLUS ADD1 SUB1) 0)
                    (ITIMES             1)
                    (PROGN
                         (SETQ BREAKCHK T)
                         NIL)))))
```

Whenever a NON-NUMERIC ARG error occurs, this entry will force the error to be caught. When the expression is evaluated, we see that only specific cases are to be treated. These are

1. The offending value is NIL, derived from

```
(NULL (CADR ERRORMESS))
```

2. If the function executed was IPLUS, ADD1, or SUB1, then substitute the value 0.
3. If the function executed was ITIMES, substitute 1.

Otherwise, break on any other occurrence of this error. This is achieved by setting BREAKCHK to T, not by returning NIL. The reason for this is that once the expression is evaluated, the value returned will be substituted in the offending statement. In this case, returning NIL will, of course, cause the error to recur.

When you use the ERRORTYPELST facility, you may examine and set the following system variables: ERRORMESS, ERRORPOS, BREAKCHK, and PRINTMSG.

ERRORMESS is a list whose CAR is the number of the error and whose CADR is the offending value. In the above example, the CADR of ERRORMESS is NIL. This yields the subcase "NON-NUMERIC ARG NIL" that we treat in the expression.

ERRORPOS is the position on the stack where the error occurred. In particular, it will be a pointer to the name of the function in which the error occurred. We obtain the name of the function by using STKNAME (see Section 30.3.2) to retrace the pointer to the PRIN1-PNAME of the function.

BREAKCHK determines whether or not a break will occur after the expression is evaluated. Note that since the expression returns a value, a break only occurs if the value causes another error. By setting BREAKCHK to T, you may force a break to occur regardless of the value returned. Conversely, setting BREAKCHK to NIL means that a break will not occur.

PRINTMSG indicates that the error message will be printed (T) or not (NIL). To force this action, you should include a SETQ expression in the ⟨expression⟩ that explicitly sets PRINTMSG.

18.2.2 An Example of ERRORTYPELST Usage

The ERRORTYPELST mechanism allows you to intercept the handling of an error and determine its disposition. You may decide to handle it yourself or allow the system to proceed with the standard processing. If you you want to handle an error yourself, you must place an entry on the ERRORTYPELST.

A common error that causes significant problems to many INTERLISP programmers is the case where a file will not open. Usually, this error arises because the file does not exist in the current directory (although on certain implementations, you may not be able to open it if it is already opened by another user).

The LISPUSERS package, FILEWONTOPEN, adds an entry to ERRORTYPELST to handle the problem where a file will not open. The entry is defined as follows:

```
(ADDTOVAR
    ERRORTYPELST
    (9
        (PROG NIL
        loop
            (AND
                (SELECTQ (STKNAME ERRORPOS)
                    (INFILE   NIL)
                    ((OUTFILE DRIBBLE) T)
                    (OPENFILE
                        (SELECTQ (STKARG 3
                        ERRORPOS)
```

```
                    (OLD NIL)
                    (NEW T)
                    (SELECTQ (STKARG 2
                    ERRORPOS)
                    ((INPUT BOTH) NIL)
                    (OUTPUT T)
                    NIL)))
          ((OPENF *PROG*LAM)
               (STKNTH -1 ERRORPOS ERRORPOS)
               (GO loop))
          NIL)
     (NOT (FILENAMEFIELD (CADR ERRORMESS)
                    'VERSION))
     (RETURN
          (PACKFILENAME
               'VERSION
               (ADD1
                    (FILENAMEFIELD
                       (OUTFILEP (CADR
                       ERRORMESS))
                       'VERSION))
               'BODY
               (CADR ERRORMESS))))))
```

18.3 CATCHING ERRORS IN A COMPUTATION

In planning for errors, you must be able to evaluate the expression that could cause the error. **ERRORSET** allows you to evaluate an expression and return a value that specifies whether an error occurred or not.

The generic format for invoking ERRORSET is

Function:	ERRORSET
# Arguments:	2
Arguments:	1) an S-expression to be evaluated, EXPRESSION 2) a flag, FLAG
Value:	Either a list containing the value of the evaluated S-expression or NIL.

ERRORSET is a LAMBDA-type function. Thus, its arguments are evaluated before passing them to ERRORSET. That is, ERRORSET performs (effectively) (EVAL (EVAL EXPRESSION)). The (EVAL EXPRESSION) is performed before ERRORSET is entered. If no error occurs in the evaluation of the S-expression, the value of ERRORSET is a list containing the value obtained by

evaluating the S-expression. Otherwise, ERRORSET returns NIL. Note that if the value of the S-expression is NIL, ERRORSET returns the value (NIL). Thus, to retrieve the value of EXPRESSION, you must take the CAR of the value returned by ERRORSET.

ERRORSET evaluates EXPRESSION via (EVAL EXPRESSION). If no error occurs as the result of evaluating EXPRESSION, ERRORSET returns the value of the computation as the sole element of a list. For example,

```
←(SETQ CITIES
      (LIST 'SAN-FRANCISCO 'NEW-YORK 'BALTIMORE))
(SAN-FRANCISCO NEW-YORK BALTIMORE)

←(ERRORSET '(CAR CITIES))
(SAN-FRANCISCO)
```

Note that I have QUOTEd the expression (CAR CITIES) because ERRORSET applies EVAL to it. Without quoting, we would have

```
←(ERRORSET (CAR CITIES) T)
UNBOUND ATOM
SAN-FRANCISCO [in ERRORSET]
NIL
```

If an error occurs during the evaluation of EXPRESSION, ERRORSET returns NIL.

```
←(ERRORSET '(IPLUS 'X 128))
NIL
```

where the obvious error is the usage of 'X in the IPLUS expression. Executing the same statement with FLAG equal to T yields

```
←(ERRORSET '(IPLUS 'X 128) T)
NON-NUMERIC ARG
X
NIL
```

FLAG determines whether or not any error message resulting from the evaluation of EXPRESSION will be printed. It may take the following values:

1. T, whence the error message is printed.
2. NIL, whence no error messages are printed. However, if NLSETQGAG is NIL, error messages are printed regardless of the value of FLAG.
3. INTERNAL, whence this invocation is ignored for purposes of determining whether or not to print an error message. ERRORSET was called

from within INTERLISP as an error prevention mechanism. The error will be handled when it is detected, so there is no reason to notify the user.

4. NOBREAK, whence no break will occur. However, a break occurs if the error occurs more than HELPDEPTH levels below the errorset.

18.3.1 Alternative Forms of ERRORSET

Two alternative forms of ERRORSET allow you to use prespecified values for FLAG. **ERSETQ** and **NLSETQ** take the form

```
Function:       ERSETQ
                NLSETQ

# Arguments:    1

Arguments:      1) an S-expression, EXPRESSION

Value:          The value produced by evaluating
                EXPRESSION; otherwise, NIL.
```

Both ERSETQ and NLSETQ are NLAMBDA functions, so their arguments are not evaluated before the function is invoked.

Consider the following example:

```
←(ERSETQ (ITIMES a b))
```

is equivalent to

```
←(ERRORSET (QUOTE (ITIMES a b)) T)
```

Note that we do not QUOTE arguments to ERSETQ and NLSETQ.

Defining ERSETQ and NLSETQ
We might define them as follows:

```
(DEFINEQ
     (ersetq
          (NLAMBDA (expression)
               (ERRORSET expression T)
     )))

(DEFINEQ
     (nlsetq
          (NLAMBDA (expression)
```

```
                (ERRORSET expression NIL)
    )))
```

The value of FLAG determines whether or not error messages are printed when ERRORSET is called. This value may be superseded by the value of NLSETQGAG. Its value is initially T, meaning don't print error messages. By setting NLSETQGAG to NIL, all error messages will be printed regardless of the value of FLAG. The effect of NLSETQGAG is to force INTERLISP to treat all NLSETQs as ERSETQs.

18.3.2 Checking for an End of File

In INTERLISP/370, the function READC does not return an indication of an end-of-file condition. Rather, it causes an error to occur. However, you may check for an end-of-file condition using ERRORSET and PEEKC. A function that might be defined is shown below:

```
(DEFINEQ
    (check-end-of-file (file-name)
        (COND
            ((ERSETQ (PEEKC file-name)) 'end-of-file))
    ))
```

PEEKC (see Section 14.2.7) "looks-ahead" in the read buffer to see what the next character is. If there is no character in the buffer and the system cannot fill the buffer with more data, an end-of-file condition results. BY applying ERSETQ, if the end-of-file condition occurs, CHECK-END-OF-FILE will return NIL. Otherwise, it returns a list consisting of the single character that would normally be returned by PEEKC.

18.4 TERMINAL-INITIATED BREAKS

Most errors and breaks occur within functions due to logic errors or because you have explicitly invoked BREAK from within a function. You may also initiate a break from your terminal. There are several reasons for breaking a program from your terminal:

1. You notice the program has entered an infinite loop that is performing the same function over and over again.
2. Your program has failed to display the expected output and you desire to determine where it is in the computation.
3. You will attempt to debug your program which has produced a noticeable error in its execution.
4. You are attempting to determine what a program does given little or no documentation (either externally or internally) and perhaps no access to the source code.

You may initiate a break from your terminal by typing an *interrupt character*. The interrupt characters differ for the particular implementation of INTERLISP. Consult the IRM to determine the interrupt characters for your implementation.

You may disable or redefine existing interrupt characters or define new interrupt characters. An interrupt character is associated with an *interrupt channel*. INTERLISP-10 has nine predefined interrupt channels:

Channel	*Interrupt Type*
CTRL-D	RESET
CTRL-E	ERROR
CTRL-B	BREAK
CTRL-H	HELP
CTRL-P	PRINTLEVEL
CTRL-T	CONTROL-T (timing statistics)
⟨DEL⟩	RUBOUT (character deletion)
CTRL-S	STORAGE
CTRL-O	OUTPUTBUFFER

INTERLISP-D does not support the STORAGE and OUTPUTBUFFER interrupt channels.

An interrupt may be either hard or soft. A hard interrupt takes place as soon as it is typed. A hard interrupt always forces the system out of the function it is currently executing. The stack is unwound to the last function call or to the top level if a CTRL-D is typed in. For example, CTRL-E and CTRL-D are hard interrupts. Soft interrupts do not occur until the next function call. Soft interrupts may be safely "continued from" without loss of data or control in the computation. CTRL-T is an example of a soft interrupt.

A hard interrupt is implemented by generating error 43. The corresponding expression from USERINTERRUPTS is retrieved once ERRORX is entered by the system.

A soft interrupt is implemented by calling INTERRUPT with an appropriate third argument. The corresponding expression is retrieved from USERINTERRUPTS. When a soft interrupt character is typed at your terminal, INTERLISP clears and saves the input buffer, and rings the bell to alert you that it has received the interrupt. The interrupt expression is evaluated. Depending on the result, the input buffers are restored and the computation proceeds.

In either case, if the character you have typed in is defined as an interrupt character, but no definition is found on USERINTERRUPTS, INTERLISP generates an error message "UNDEFINED USER INTERRUPT" and forces a break.

18.5 TYPES OF ERRORS

INTERLISP provides error indications for over fifty types of errors. Each error is indicated by a numeric code and an associated message. The numeric code is returned by ERRORN. The alphanumeric string will be printed by ERRORMESS. Some errors are implementation dependent, and thus are undefined in other versions of INTERLISP.

Error 0: System Trap Error
Error code 0 corresponds to a "SYSTEM ERROR". This error also occurs in the INTERLISP-10 system, where it is displayed as "JSYS ERROR". It is undefined in INTERLISP/370.

For INTERLISP-10, JSYS errors occur if you attempt to execute an operating system function with the wrong code or parameter setup. It may also occur if you are programming directly in ASSEMBLE code. Usually it occurs because of addressing problems due to smashed data structures.

The IRM advises that you abandon (e.g., reload) a system that exhibits this behavior.

Error 1: Unused Error
This error code is not used in any of the current INTERLISP implementations.

Error 2: Stack Overflow
Error code 2 corresponds to a "STACK OVERFLOW". This error usually occurs as the result of a runaway computation. The computation calls too many nested functions or binds to many variables in the functions that it calls.

One way to invoke this error in INTERLISP/370 is by attempting to trace EVAL. The TRACE function (see Section 20.1.1) calls EVAL internally to evaluate the expression which is its argument.

Another way is to begin a recursive computation where the number of levels of recursion exceeds the stack depth. For example, trying to count the number of cells in a circular list using a recursive function that operates on successive tails of the list.

Error 3: Illegal Return
Error code 3 corresponds to an "ILLEGAL RETURN". This error results when a program attempts to execute a RETURN expression while not executing within a PROG form.

Note that a call to RETURN may occur in a function called from within a PROG in another function.

Error 4: Argument Not a List
Error code 4 corresponds to "ARG NOT LIST". This error results when the function invoked expects a list as its argument.

One common error is passing an atom as the first argument of RPLACA or RPLACD rather than a list. For example,

```
←(RPLACA 'X (LAST X))
ARG NOT LIST
X
```

Error 5: File System Error
Error code 5 corresponds to "HARD DISK ERROR". This error code is currently defined for INTERLISP-D but not for INTERLISP-10 or INTERLISP/370. This error indicates that something is wrong with the integral disk used in Xerox 11xx systems. You should consult a Field Engineer if this error occurs.

Error 6: Setting NIL
Error code 6 corresponds to "ATTEMPT TO SET NIL OR T". This error results when the first argument of SET or SETQ is NIL. Usually, you will encounter this error when INTERLISP attempts to evaluate an argument to SET whose value is NIL.

Error 7: Attempting to Replace NIL
Error code 7 corresponds to "ATTEMPT TO RPLAC NIL". It occurs when you attempt to RPLACA or RPLACD NIL with something other than NIL.

Error 8: Invalid Goto
Error code 8 corresponds to "UNDEFINED OR ILLEGAL GO". This error occurs when a program attempts to execute (GO ⟨label⟩) while not within a PROG form or when ⟨label⟩ is undefined within the PROG.

Error 9: Unable to Open a File
Error code 9 corresponds to "FILE WON'T OPEN". This error occurs when a program executes one of the File Package functions that open a file. Usually, the file does not exist in the current directory.

Error 10: Non-numeric Argument
Error code 10 occurs when a numeric function expects a number as an argument. For example,

```
←(IPLUS 'five 10)
NON-NUMERIC ARG
FIVE
```

Error 11: Atom Too Long
Error code 11 corresponds to the error "ATOM TOO LONG". This error occurs when you attempt to a create an atom with a PRIN1-PNAME whose length is greater than the maximum number of characters allowed by your implementation. Usually, it will occur if you type an atom name that is too long, if you try to create an atom name via PACK, or you read an atom name from a file. The first and last cases are generally due to typing errors and may be easily corrected.

Error 12: Atom Hash Table Filled
Error code 12 corresponds to "ATOM HASH TABLE FULL". The atom hash table is used to store pointers to each of the atoms in the system in order to speed reference to their particular storage locations. It is possible to exceed the size of the atom hash table. Usually, the atom hash table starts with an initial size and expands automatically until the maximum allowable size is reached. For INTERLISP-10, this limit is approximately 32,767 atoms.

Error 13: File Not Open
Error code 13 corresponds to "FILE NOT OPEN". Most input and output functions as well as many of the File Package functions expect the file that they will operate on to be open. If the file is not open, this error is generated.

Error 14: Argument Not a Literal Atom
Error code 14 corresponds to "ARG NOT LITATOM". Many INTERLISP primitives expect an atom as an argument. This error is generated when the argument is determined to be something other than a literal atom.

Error 15: Too Many Files Open
Error code 15 corresponds to "TOO MANY FILES OPEN". Different implementations set a limit on the number of files that your program may have open simultaneously. Usually, this number is about 32. Because the terminal is always open and available for input/output, it is not included in the open file count.

Error 16: End of File Encountered
Error code 16 corresponds to "END OF FILE". This error occurs when an input function attempts to read past the end of a file. After the error occurs, the file is closed.

This error can be very disconcerting to most users. It probably should be handled better, i.e., some indication of an end-of-file condition being returned without causing an error.

You may process this error by placing an entry on ERRORTYPELST to intercept it. Your code may or may not close the file, but should return some indication of an end-of-file condition. The IRM suggests that you return "]" as an indication meaning the end of a list.

Error 17: Error
Error code 17 corresponds to "ERROR". It is produced when the function ERROR is called with one or more messages appropriate to the particular function that generated the error. You may also want to use this error code for errors occurring within your programs.

Error 18: CTRL-B condition
Error code 18 corresponds to "BREAK". This error occurs when you type a CTRL-B during the execution of a program. Your computation is placed under

the control of the BREAK package (see Chapter 20) at the current execution point.

Error 19: Illegal Stack Argument
Error code 19 corresponds to "ILLEGAL STACK ARG". This error occurs when a stack manipulation function is given an argument other than a stack position. The most common case occurs when you attempt to specify a stack position via a function name which cannot be found in the stack. See Chapter 30 for a discussion of the Stack Functions.

Error 20: Evaluation Fault
Error code 20 corresponds to "FAULT EVAL". This occurs while INTERLISP is being loaded (bootstrapped) into memory. Once FAULTEVAL is defined, this error should not occur.

Error 21: Array Space Exceeded
Error code 21 corresponds to "ARRAYS FULL". As mentioned earlier, INTERLISP divides available memory into a number of pools corresponding to the basic datatypes that it supports. When array space is exceeded, INTERLISP first attempts a garbage collection on the array space. If this does not produce enough array space, then this error message is generated.

This error usually occurs because you have created too many arrays or tried to create an array with unusually large dimensions. The latter case occurs when you specify arguments to ARRAY which compute the size of the array. If your arguments produce erroneous dimension information, you will receive this error.

Error 22: File System Space Exceeded
Error code 22 corresponds to "FILE SYSTEM RESOURCES EXCEEDED". This error occurs when the number of files that you have written exceeds the current file space. For INTERLISP-10, this may be attempting to exceed the limits imposed upon your directory concerning number of files, file blocks, or size. In INTERLISP-D, it often occurs when you attempt to exceed the capacity of the integral disk in the Xerox 11xx machines.

Error 23: Non-existent File
Error code 23 corresponds to "FILE NOT FOUND". This error occurs when you attempt to open a file which does not exist in the specified directory. It may also occur if the file name is ambiguous (i.e., when several files have the same file name but different extensions).

ERRORTYPELST is initialized to handle this error as follows:

```
(SETQ ERRORTYPELST
      '(23
                (SPELLFILE (CADR ERRORMESS))))
```

where SPELLFILE will search alternative directories or perform spelling correction on the directory or file name to determine the proper file.

Error 24: Damaged SYSOUT File
Error code 24 corresponds to "BAD SYSOUT FILE". This error occurs when the date of the sysout file (see Section 16.9.1) do not correspond to the data on which the current version of INTERLISP was made. It also occurs if you attempt to load a file via SYSIN which is not a sysout file.

Error 25: Bad CDR Argument
Error code 25 corresponds to "UNUSUAL CDR ARG LIST". This error occurs when an expression ends in a non-list other than NIL. For example,

```
←(CONS T . 3)
UNUSUAL CDR ARG LIST
3
```

Error 26: Hash Table Space Exceeded
Error code 26 corresponds to "HASH TABLE FULL". This error occurs when you attempt to hash more items into a hash table than it has room to store. See Section 11.3.

Error 27: Illegal Argument
Error code 27 corresponds to "ILLEGAL ARGUMENT". This error is displayed when INTERLISP does not have a specific error message to produce. It is used by many functions within INTERLISP. Actually, errors of this sort should be better handled, but no one has taken the time to rewrite the internal code to categorize them.

Error 28: Non-Array Argument
Error code 28 corresponds to "ARG NOT ARRAY". This error occurs when an array function is given an argument which is not an array address. See Section 11.1.

Error 29: Illegal Block
Error code 29 corresponds to "ILLEGAL OR IMPOSSIBLE BLOCK". This error occurs in INTERLISP-10 when you attempt to get or release an additional block of memory.

Error 30: Already Released Stack Pointer
Error code 30 corresponds to "STACK PTR HAS BEEN RELEASED". This error occurs when you give a stack function a stack pointer that is no longer defined, because it has been released. See Chapter 31.

Error 31: Memory Space Exceeded
Error code 31 corresponds to "STORAGE FULL". This error occurs when the available memory space has been entirely allocated. Usually, it occurs after a garbage collection when the garbage collector has not been able to find any unallocated space. This error may occur (in lengthy computations) if you allocate space for various objects, but fail to retain their addresses via SETQ. The space is consumed, but disappears from your virtual memory. You cannot address if you do not retain a pointer to it, and it cannot be garbage collected for the same reason.

This error will also occur if your computation is too large for the available memory space. For example, the Xerox 1108/1186 machines have a virtual memory space of 32 Mbytes. It is fairly easy for large computations to exceed this memory space after some interval. When an error of this sort occurs, you may be placed in RAID.

Error 32: Incorrect Object Types
Error code 32 corresponds to "ATTEMPT TO USE ITEM OF INCORRECT TYPE". This error occurs in Record Package functions and in functions associated with accessing user-defined datatypes (see Chapter 27). This error is generated if the type of change to be made to a field of a user datatype does not correspond to the datatype of the object.

Error 33: Illegal Datatype Number
Error code 33 corresponds to "ILLEGAL DATATYPE NUMBER". This error occurs when you present a user-defined datatype function with an invalid field specifier. See Chapter 27.

Error 34: Datatype Space Exceeded
Error code 34 corresponds to "DATATYPES FULL". The memory pool available in which to allocate descriptions of user-defined datatypes is limited. When this pool is full, this error will be generated.

Error 35: Binding NIL or T
Error code 35 corresponds to "ATTEMPT TO BIND NIL OR T". This error occurs in PROG and LAMBDA expressions where you declare NIL or T as a dummy variable and attempt to assign it a value. For example,

```
←(PROG ((NIL ⟨expression⟩)) ...)
ATTEMPT TO BIND NIL OR T
NIL
```

Error 36: User Interrupt Characters Exceeded
Error code 36 corresponds to "TOO MANY USER INTERRUPT CHARACTERS". INTERLISP currently provides for nine user interrupt channels. Attempting to enable an additional user interrupt character will cause this error.

Error 37: Read Macro Error
Error code 37 corresponds to "READ-MACRO CONTEXT ERROR". This error occurs when READ is executed within a read macro function and the next token is) or]. See Section 14.4.

Error 38: Illegal Read Table
Error code 38 corresponds to "ILLEGAL READTABLE". This error occurs when you provide an illegal address to one of the read table functions (see Section 14.5).

Error 39: Illegal Terminal Table
Error code 39 corresponds to "ILLEGAL TERMINAL TABLE". This error occurs when you provide an illegal address to one of the terminal table functions (see Section 15.5).

Error 40: Swap Block Too Large
Error code 40 corresponds to "SWAPBLOCK TOO BIG FOR BUFFER". This error occurs in INTERLISP-10 when you attempt to swap in a function or an array which exceeds the size of the swapping buffer. Consult the IRM (page 22.26) for further details.

Error 41: Protection Violation
Error code 41 corresponds to "PROTECTION VIOLATION". This error occurs in INTERLISP-10 when you attempt to open a file that you do not have clearance to access or you attempt to reference an unassigned device. Consult the host operating system manual.

Error 42: Illegal File Name
Error code 42 corresponds to "BAD FILE NAME". This error occurs when you provide a badly formed file name to one of the File Package functions. The error will be generated by the host operating system when it attempts to resolve the file name in the disk directory.

Error 43: User Break
Error code 43 corresponds to "USER BREAK". This error occurs when you generate a hard user interrupt character, e.g., CTRL-B, CTRL-D, etc.

Error 44: Unbound Atom
Error code 44 corresponds to "UNBOUND ATOM". This error occurs when you attempt to reference an atom which has not been defined. The atom is created, but its value cell contains NOBIND. For example (assuming X was previously defined),

```
←(SETQ X 'NOBIND)
(X reset)
NOBIND
```

```
←X
UNBOUND ATOM
X
```

The atom has no stack binding due to a PROG or LAMBDA expression, nor a top level value. The atom which generated the error can be retrieved via (CADR ERRORMESS).

Error 45: Undefined Expression CAR
Error code 45 corresponds to "UNDEFINED CAR OF FORM". This error occurs when EVAL attempts to evaluate an expression whose CAR is not a valid function name. The atom may exist, but it has no definition in the function cell.

```
←(EVAL (X Y))
UNDEFINED CAR OF FORM
X
```

Error 46: Undefined Function
Error code 46 corresponds to "UNDEFINED FUNCTION". This error occurs if APPLY is given as its first argument an atom which does not have a function definition. For example,

```
←(APPLY 'X (LIST 'A 'B 'C))
UNDEFINED FUNCTION
X
```

Error 47: CTRL-E Error
Error code 47 corresponds to "CONTROL-E". This error is generated when you type a CTRL-E at your terminal.

Error 48: Floating Point Underflow
Error code 48 corresponds to "FLOATING UNDERFLOW". This error is generated in INTERLISP-D only during a floating point operation.

Error 49: Floating Point Overflow
Error code 49 corresponds to "FLOATING OVERFLOW". This error is generated in INTERLISP-D only during a floating point operation.

Error 50: Overflow
Error code 50 corresponds to "OVERFLOW". This error is generated in INTERLISP-D only during an integer operation.

Error 51: Illegal Hash Array
Error code 51 corresponds to "ARG NOT HARRAY". This error is generated in INTERLISP-D only when a hash array function is given an address which does

not point to a hash array. Note that INTERLISP-D allocates separate memory pools for arrays and hash arrays and can thus detect the distinction easily.

Error 52: Too Many Arguments
Error code 52 corresponds to "TOO MANY ARGUMENTS". This error is generated in INTERLISP-D only when the evaluation function detects that a LAMBDA spread or nospread, or an NLAMBDA spread function has been given too many arguments. This error is generated by the Xerox 11xx machine microcode.

18.6 ERROR HANDLING FUNCTIONS

INTERLISP provides a powerful and flexible error handling environment. It is usually possible for you to write programs that can recover from almost any type of error. To do so, however, you must be able to analyze the error occurrence. When it is not possible to recover from an error, you must be able to provide sufficient information to enable the user to diagnose the problem and correct it. The functions described in this section are used to provide various types of information about the occurrence of errors.

18.6.1 Printing Error Messages

Several functions allow you to print standard error messages that are augmented with additional information.

ERROR prints one or two error messages and causes a break to occur. It takes the form

```
Function:       ERROR
# Arguments:    3
Arguments:      1) a message, MESSAGE1
                2) a message, MESSAGE2
                3) a break flag, NOBREAKFLAG
Value:          NIL.
```

When ERROR is called, it prints MESSAGE1 and MESSAGE2 as follows:

1. MESSAGE1 is printed using PRIN1. If MESSAGE1 is an atom, a space is printed, and then MESSAGE2 is printed using either PRIN1, if it is a string, or PRINT, if it is anything else.
2. If MESSAGE1 is not an atom, then after it is printed using PRIN1, a carriage return is printed, and then MESSAGE2 as described above.
3. If MESSAGE1 and MESSAGE2 are both NIL, the message will simply be ERROR.

If NOBREAK is T, ERROR prints its messages and calls ERROR! (see below). Otherwise, it executes

```
(ERRORX 17 (CONS MESS1 MESS2))
```

which leaves the decision as to whether to print any messages and cause a break up to BREAKCHECK.

A Definition for ERROR

We might define ERROR as follows:

```
(DEFINEQ
    (error (message1 message2 nobreakflag)
        (COND
            (nobreakflag
                (*
                    Print the error messages, but do
                    not break the computation here.
                )
                (SETERRORN 17
                           (CONS message1 message2))
                (ERRORMESS1 message1 message2 'ERROR)
                (ERRORB))
            (T
                (ERRORX
                    (LIST 17
                          (CONS message1
                          message2)))))
    ))
```

ERRORB is an internal function that is implementation dependent.

Printing Help Messages

An alternative function is **HELP**. It takes the form

```
Function:       HELP
# Arguments:    2
Arguments:      1) a message, MESSAGE1
                2) a message, MESSAGE2
Value:          The value returned from the breakpoint.
```

HELP prints its messages like ERROR and then calls BREAK1 (see Section 20.3.3). If MESS1 and MESS2 are both NIL, HELP merely prints HELP! before calling BREAK1. For example,

```
←(HELP "An example of HELPing")
AN EXAMPLE OF HELPING
NIL

(HELP broken)
:
```

A Definition of HELP

We might define HELP as follows:

```
(DEFINEQ
     (help (message1 message2)
           (ERRORMESS1 message1 message2 'HELP!)
           (BREAK1 (ERROR "??" 'G T)
                   T
                   HELP)
     ))
```

Printing a Warning to the User

SHOULDNT calls HELP with the message "Shouldn't Happen!". It takes the form

```
Function:       SHOULDNT

# Arguments:    0

Arguments:      N/A

Value:          No value returned.
```

HELP and SHOULDNT provide you with different ways of signaling unusual conditions in your program. HELP indicates that the program has entered a region that it normally would not enter at this stage of the computation. A good use is to signal an erroneous data value which you might allow the user to correct. On the other hand, SHOULDNT indicates that the program should never have encountered the situation that it finds itself in.

```
←(SHOULDNT)
"Shouldn't Happen"

(HELP broken)
:STOP
                              (Note: no value returned!)
←
```

Generalized Message Printing Functions

ERRORMESS is a generalized message printing routine that prints an error message corresponding to an ERRORN form. It takes the form

```
Function:       ERRORMESS
# Arguments:    1
Argument:       1) an ERRORN form, ERRORFORM
Value:          NIL.
```

ERRORMESS prints a message corresponding to the ERRORN form. For example,

```
←(ERRORMESS '(12 T))
ATOM HASH TABLE FULL
T
NIL
```

A Definition for ERRORMESS

We might define ERRORMESS as follows:

```
(DEFINEQ
     (errormess (error-expression)
          (COND
               ((NULL error-expression)
                    (*
                        Retrieve error form corresponding
                        to the last error that occurred.
                    )
                    (SETQ error-expression (ERRORN))))
          (COND
               ((EQUAL (CAR error-expression 17)
                    (*
                         17 signals a general error code.
                    )
                    (SETQ error-expression
                          (CADR error-expression))
                    (ERRORMESS1 (CAR error-expression)
                                (CDR error-expression)))
               (T
                    (AND LISPXHISTORY
                         (LISPXPUT 'ERROR
                                   (CADR error-
                                   expression)))
                    (ERRORM error-expression)))
     ))
```

ERRORMESS1

The workhorse function for printing error messages for many of these functions is ERRORMESS1. It takes the form

```
Function:       ERRORMESS1
# Arguments:    3
Arguments:      1) a message, MESSAGE1
                2) a message, MESSAGE2
                3) a message, MESSAGE3
Value:          NIL.
```

ERRORMESS1 prints the messages for HELP and BREAK functions. We might define ERRORMESS1 as follows:

```
(DEFINEQ
    (errormess1 (message1 message2 message3)
        (PROG (a-message)
              (COND
                  ((AND
                       (NULL message1)
                       (NULL message2))
                       (*
                            If MESSAGE1 and MESSAGE2
                            are null, print MESSAGE3
                            and return.
                       )
                       (LISPXPRINT message3 T)
                       (RETURN)))
              (LISXPRIN1 message1 T)
              (COND
                  ((OR
                       (ATOM message1)
                       (STRINGP message2))
                       (*
                           If MESSAGE1 is an atom or
                           MESSAGE2 is a string, print
                           a space and MESSAGE2.
                       )
                       (LISPXSPACES 1 T))
                  (T
                       (LISPXTERPRI T)))
              (SETQ a-message message1)
              (COND
                  ((STRINGP message2)
```

```
                    (LISPXPRIN1 message2 T)
                    (LISPXTERPRI T))
               (T
                    (SETQ a-message message2)
                    (LISPXPRINT message2 T)))
          (AND
               LISPXHISTORY
               (LISPXPUT 'ERROR a-message))
))
```

Note that A-MESSAGE records the proper message for insertion into the history list.

18.6.2 Returning from Errors

There are two ways to return from an error: ERROR! and RESET.

ERROR! takes the form

Function:	ERROR!
# Arguments:	0
Arguments:	N/A
Value:	?? (not to be trusted)

ERROR! operates like a programmable CTRL-E. When executed, it immediately returns from the most recent ERRORSET or RESETLST (see Section 25.7).

Resetting the System State

RESET operates like a programmable CTRL-D. It forces an immediate return to the top level of INTERLISP. It takes the form

Function:	RESET
# Arguments:	0
Arguments:	N/A
Value:	NIL

RESET clears the stack of all frames associated with the previous computation.

```
←(RESET)
```

RESET is often issued from within the Break Package when one discovers that the environment has become too entangled to decipher.

Note, however, that issuing a RESET does not re-initialize your environment. Thus, any objects that you have created in the computation thus far will remain defined. This "detritus" can affect the re-execution of your program, particularly if it expects to find certain things uninitialized or having certain values.

18.6.3 Obtaining Information about Errors

ERRORN returns a form that describes the last error that occurred. It takes the form

```
Function:        ERRORN
# Arguments:     0
Arguments:       N/A
Value:           An expression (⟨error #) . x).
```

The error number is the index of the last error that occurred. X is the expression that was printed or would have been printed after the error message (e.g., the value of MESSAGE1).

Defining New Error Messages

You may define new error messages using the **SETERRORN,** which takes the form

```
Function:        SETERRORN
# Arguments:     2
Arguments:       1) an error number, NUMBER
                 2) an error message, MESSAGE
Value:           The error number.
```

SETERRORN may be used to redefine any of the existing system error messages (usually with longer, more detailed messages) or to define a new error message. INTERLISP-D supplies over fifty error messages. You may use numbers greater than 52 for you own error messages.

The best way to initialize error messages for a program is to place SETERRORN calls in the P command list of the File Package commands. These calls are evaluated when the file is loaded, and so are part of the initialization of your environment.

If you define new error messages, I recommend that you categorize them according to the type, location, and severity of error. You should also assign you error messages numbers beginning with 200 to avoid possible future conflict with INTERLISP error messages.

Obtaining the Error Message

ERRORSTRING returns the string which represents the message associated with an error number. It takes the form

```
Function:       ERRORSTRING
# Arguments:    1
Arguments:      1) an error number, NUMBER
Value:          The string, if any, associated with the
                indicated error number.
```

ERRORSTRING is primarily used to allow you to obtain an error message which you may then edit via the string handling functions before printing via ERROR.

```
←(ERRORSTRING 10)
"NON-NUMERIC ARG"
```

ERRORSTRING is used when you want to tailor the error messages more specifically to conditions that arise within your program. Since you may intercept the printing of error messages, you may obtain the basic string and edit it to be more informative for the user.

18.6.4 Entering the Error Routines

The general entry point to the error routines is **ERRORX**, which takes the form

```
Function:       ERRORX
# Arguments:    1
Argument:       1) an error message form, ERRORFORM
Value:          The value returned by the exit from the
                error routines.
```

If ERRORFORM is NIL, ERRORN is used to determine the error. Otherwise, ERRORFORM should have the form:

```
(⟨number⟩ . ⟨message⟩)
```

whence ERRORX calls SETERRORN with the CAR and CDR of the form to (re)define the error message.

ERRORX calls BREAKCHECK to determine whether an error should occur or not. If no break occurs, it prints the message and returns from the last ERRORSET.

A Definition for ERRORX

We might define ERRORX as follows:

```
(DEFINEQ
     (errorx (errorform)
          (ERRORX2
               (COND
                    (errorform
                         (*
                              Caller has provided error
                              number and message.
                         )
                         (SETERRORN (CAR errorform)
                                    (CADR errorform))
                         errorform)
                    (T
                         (*
                               Use the error number and
                               message determined by the
                               system.
                         )
                         (ERRORN)))
               (STKNTH -1 'ERRORX))
     ))
```

ERRORX2 is the workhorse that actually performs all of the functions associated with handling errors. We might define it as follows:

```
(DEFINEQ
   (errorx2 (error-message error-position)
      (PROG (culprit array-size array-address esgag
            frame-name fn-type breakchk)
            (COND
               ((AND
                    (EQ (CAR error-message) 26)
                    (SETQ culprit (CADR error-message))
                    (LISTP culprit))
                    (*
                         Error was a hash table overflow.
                    )
                    (SETQ array-size
                          (ARRAYSIZE (CAR culprit)))
                    (SETQ array-address (CDR culprit))
                    (SETQ array-address
                          (HARRAY
```

```
                        (COND
                           ((NULL array-address)
                               (FTIMES array-size
                               1.5))
                           ((FLOATP array-address)
                               (FTIMES array-size
                                       array-
                                       address))
                           ((NUMBERP array-address)
                               (IPLUS array-size
                                      array-
                                      address))
                           (T
                               (GO top)))))
                  (RPLACA culprit
                          (REHASH (CAR culprit)
                                  array-address)))
                  (RETURN culprit)))
        (SETQ breakchk
              (BREAKCHECK error-position
                          (CAR error-message)))
top
        (COND
           ((AND
                 ERRORTYPELST
                 (SETQ culprit
                       (ASSOC (CAR error-message)
                              ERRORTYPELST))
                 (SETQ culprit (EVAL (CADR culprit))))
                 (*
                     If there are entries on
                     ERRORTYPELST and the error is
                     represented by one of those
                     entries, then evaluate the form
                     found there to handle the error.
                 )
                 (RETEVAL
                    error-position
                    (SUBPAIR
                        '(FN ARGS)
                        (LIST
                           (LINKSTKNAME
                              (STKNAME error-position)
                              error-position)
                           (SUBST
```

```
                                    culprit
                                    (CADR error-message)
                                    (STKARGS error-
                                    position)))
                              '(APPLY 'FN 'ARGS))))
            ((NULL esgag)
                 (*
                      Do not suppress the error
                      message
                 )
                 (ERRORMESS error-message)))
errorb
       (COND
          ((NULL breakchk)
                 (*
                      The error will occur at the
                      actual position in the code.
                 )
                 (RETEVAL
                      (STKNTH 1 error-position)
                      'ERRORB))))
       (SETQ frame-name (STKNAME error-position))
       (COND
          ((NOT (LITATOM frame-name))
                 (SETQ frame-name
                       (LINKSTKNAME frame-name
                                    error-position))))
       (COND
          ((EQ (CAR error-message) 10)
                 (*
                      Non-numeric argument to
                      arithmetic function.
                 )
                 (LISPXPRIN1 'IN T)
                 (LISPXPRINT frame-name T)
                 (SETQ result
                       '(BREAK1
                           (ERROR '? 'G T)
                           T
                           NIL
                           NIL
                           ERRORX)))
                 (GO exit))
       (SETQ fn-type
             (SELECTQ (FNTYP frame-name)
```

```
                ((SUBR* FSUBR FSUBR*)
                   '(ERROR 'CANT 'G T))
                ((EXPR FEXPR EXPR* FEXPR*)
                   (CONS 'PROGN
                         (CDDR (GETD frame-name))))
                (NIL
                   (CONS 'PROGN
                         (CAR
                           (VAG
                             (ADD1
                               (LOC frame-name))))))
                (SUBR
                   (RPLACD (STKARG 1 error-position)
                           'U)
                   (RPLACD (STKARG 2 error-position)
                           'V)
                   (CONS frame-name '(U V)))
                (SAVED1 fn-type
                        frame-name
                        (ARGLIST frame-name))))
        (SETQ result
              (LIST 'RETFROM
                    (LIST (QUOTE QUOTE) error-
                     position)
                    (LIST 'BREAK1
                          culprit
                          T
                          frame-name
                          'ERRORX)))
exit
        (RETEVAL
           (STKNTH 1 error-position)
           result))
))
```

The stack functions are described in Chapter 31. RETEVAL, described in Section 30.8, allows you to return from a stack frame with a value without actually completing the computation that was to be performed by the function represented at the stack frame. The purpose of much of the code in the above function is to define a list which actually computes the result to be returned from the frame. RETEVAL evaluates the list, places the value in the appropriate frame location, and returns to the caller.

19

The INTERLISP Editor

INTERLISP provides an integrated editing facility that understands the characteristics of all the data structures that it supports. The editor can be invoked to edit any data structure either at the top-level READ-EVAL-PRINT loop or as a function invoked from another function within a user program. When called from within a program, an optional list of commands may be passed to the editor for execution.

This editor is to be used to edit expressions from a terminal/keyboard combination. INTERLISP-D provides DEdit, a display editor, which works with the mouse and bit-mapped screens of the Xerox 11xx machines. The latter editor will be discussed in Volume 2.

The Editor is just one of a number of programming utilities that INTERLISP provides to create an integrated programming environment. This chapter and the next discuss the Editor. Subsequent chapters discuss other programming utilities.

19.1 INVOKING THE EDITOR

The Editor may be used to edit function definitions, property lists, variable values, and arbitrary expressions. It may be invoked from either the top level READ-EVAL-PRINT loop or from within a user program. When called from the top level, it seeks commands from the input file T corresponding to the user's terminal. When called from a user's program, it may be passed an optional list of commands for execution which it performs and then returns. The Editor is invoked by different functions for the modes associated with specific data structures.

19.1.1 Function Editing

You may edit a function definition by executing **EDITF**, which takes the following form

```
Function:        EDITF
                 EDITFNS

# Arguments:     1-N

Arguments:       1) a function name, FN
                 2) zero or more commands, COMMAND[1] ...
                    COMMAND[N]

Value:           The name of the function (EDITF)
                 or NIL (EDITFNS).
```

EDITF is an NLAMBDA, nospread function. You must provide a function name as the first argument. Optionally, you may pass it a sequence of commands to be executed. EDITF returns the name of the function that it has edited, if successful.

When EDITF is called, it must check several cases before editing the function.

The First Argument is an Expression

The first argument is an EXPR. EDITF calls EDITE (see below) to edit the definition using the following statement:

```
(PUTD name
      (EDITE
          (GETD 'fn)
          (LIST 'command[1] ... 'command[N])
          'fn
          'FNS))
```

Note: PUTD and GETD are described in Sections 8.3 and 8.4, respectively.

NAME may be an EXPR because it is broken or advised. Thus, several subcases must be considered:

1. If NAME really is an EXPR, editing the broken or advised function will affect the original definition. However, the structure of the function definition is changed. You are notified by a warning message so that you may position yourself properly before issuing editing commands.

```
←(DEFINEQ
    (trash (x)
        (RPLACA x NIL)))
(TRASH)

←(BREAK trash)
(TRASH)
```

```
←(EDITF trash)
Note: you are editing a broken definition.
EDIT
*PP
(LAMBDA (X)
     (BREAK1
          (PROGN (RPLACA X NIL))
                 T TRASH NIL))
```

2. If NAME is not an EXPR and has no EXPR property, the File Package looks to see if it "knows" where NAME is contained (e.g., on any noticed file). If it does, the EXPR definition of NAME is loaded onto its property list under the property EXPR. Editing continues as is in (2) below. If the File Package cannot find the source definition, a warning is printed, and editing proceeds. The latter case is possible when you advise compiled code, because you may want to edit the advice.
3. If NAME is not an EXPR, but it does have an EXPR property, NAME is unbroken or unadvised (only with your approval, since you may want to edit the advice) and proceeds as in (2).

The First Argument has an EXPR Property

If the first argument is not an expression, it may have an EXPR property (see Section 8.1). EDITF prints PROP and executes:

```
(EDITE
     (GETPROP 'fn 'EXPR)
     (LIST 'command[1] ... 'command[N])
     'fn
     'PROP)
```

When EDITE completes, several possibilities may have occurred:

1. If no changes were made, EDITE responds with the message "NOT CHANGED, SO NOT UNSAVED"

```
←(SETQ trash NIL)
NIL
←(PUTPROP trash
          'EXPR
          '(LAMBDA (x) (RPLACA x NIL)))
TRASH
←(EDITF trash)
PROP
EDIT
```

```
*ok
Not changed, so not unsaved.
```

2. If changes were made, and DNAMEFLG has the value PROP, EDITE responds with the message "CHANGED, BUT NOT UNSAVED"

```
←DNAMEFLG
PROP

←(EDITF trash)
PROP
EDIT
*PP
(LAMBDA (x) (RPLACA x NIL))
*3 P
(RPLACA x NIL)
*(REPLACE 1 WITH RPLACD)
*OK
Changed, but not unsaved.
```

3. Otherwise, EDITE responds with the message "UNSAVED" and executes UNSAVEDEF (see Section 17.5.7).

   UNSAVEDEF restores the definition of the function from its property list. The effect of the invocation of EDITE is to put the modified definition on the property list of the first argument (evaluating to a literal atom) under the property PROP. Then, UNSAVEDEF copies the definition back to the EXPR property.

   Note: INTERLISP/370 unsaves in any case.

The First Argument (as a Literal Atom) is Contained in some File Known to the File Package

The first argument is neither an expression nor does it have an EXPR property value. When evaluated, it resolves to a literal atom. The File Package determines if the atom is contained in a file that it knows. If so, it loads the EXPR definition of the atom from the file using LOADDEF (see Section 17.5.8). It then operates like (2) above.

The First Argument has a Definition

The first argument is neither an expression nor has an EXPR property value, but it does have a definition. EDITF prints ⟨name⟩ NOT EDITABLE.

```
←(SETQ trash '(RPLACA x NIL))
(RPLACA x NIL)
```

```
←(EDITF trash)
trash not editable.
```

You should use EDITV instead.

The First Argument's Value is a List

If the first argument's value is a list, EDITF assumes that you meant to call EDITV. It displays "=EDITV" and invokes EDITV for you:

```
←(SETQ sentence '(now is the time for all good men))
(NOW IS THE TIME FOR ALL GOOD MEN)
←(EDITF sentence)
=EDITV
EDIT
*P
(NOW IS THE TIME FOR ALL GOOD MEN)
```

Alternatively, if NAME has a non-NIL property list, the editor assumes you meant to call EDITP. It displays the message "=EDITP" and invokes it for you. For example,

```
←(PUTPROP sentence
          'value
          '(now is the time for all good men))
(NOW IS THE TIME FOR ALL GOOD MEN)
←(EDITF sentence)
=EDITP
EDIT
*P
(NOW IS THE TIME FOR ALL GOOD MEN)
```

The editor invokes EDITP only if the variable value is NIL; otherwise, EDITV takes precedence.

None of the Above

If none of the above cases pertains, EDITF attempts to correct the spelling of the name using the spelling list USERWORDS (see Section 22.7.2). If successful, it begins anew at (1).

Whenever a function has been successfully edited, EDITE inserts a time stamp in the form of a comment (* xxx ⟨date⟩) in the function definition. XXX are the user's initials (see INITIALS). If a time stamp already exists, only the date in the comment will be changed.

```
←(SETQ INITIALSLST (CONS 'kaisler 'shk))
(KAISLER . SHK)

←(DEFINEQ (trash (x) (RPLACA x NIL)))
(TRASH)

←(SETQ **COMMENTFLG** T)
T

←(EDITF trash)
*3 P
(RPLACA x NIL)
*(REPLACE 1 WITH SETQ)
*OK
(TRASH)

←(PP trash)
(LAMBDA (x)
     (SETQ x NIL))
```

Often, we want to perform the same set of commands on several functions. The Editor provides the function EDITFNS to accomplish this task. In this case, the value of the first argument is an S-expression that evaluates to a list of one or more function names. EDITFNS calls EDITF on each function with the optional list of commands, if any. EDITFNS always returns NIL.

FN may be an atom. If its value is not a list, but is a noticed file name, FILEFNSLST (see Section 17.7.3) is invoked to return a list of functions to be edited. FILEFNSLST extracts the functions from the FNS command associated with the File Package commands.

When EDITFNS executes, each call to EDITF will be ERRORSET protected. Thus, it continues to process the list of functions even if one element of the list causes an error. This might occur if one of the editing commands fails for an element or an element is not a function.

19.1.2 Value Editing

You edit the value of a variable by executing the function EDITV. EDITV, like EDITF, takes the following form

```
Function:       EDITV

# Arguments:    1-N

Arguments:      1) a variable name, NAME
                2-N) editor commands, COMMAND[1] ...
                     COMMAND[N]

Value:          The name of the variable.
```

EDITV is an NLAMBDA, nospread function.

Usually, EDITV is called with a single atom as its argument. It may have one or more optional arguments which are Editor commands to be performed upon the variable's value. The editor checks several different cases:

1. If NAME is an atom and has the value NOBIND, the Editor determines if it is the name of a function. If so, it assumes you meant to call EDITF, displays the message "=EDITF", and calls EDITF with the name of the function.

```
←trash
U.B.A.

←(EDITV trash)
=EDITF
EDIT
*P
(LAMBDA (X) (RPLACA X NIL))
```

   Note that TRASH may not have a value in its value cell, but may have a definition in its function definition cell.

2. If NAME is an atom and has a value, EDITV allows you to edit the value:

```
←(SETQ trash.flag 'don't)
'DON'T

←(EDITV trash.flag)
DON'T NOT EDITABLE.
```

   because an atomic value cannot be changed, but

```
←(SETQ phrase '(don't rain on my parade))
(DON'T RAIN ON MY PARADE)

←(EDITV phrase)
EDIT
*P
(DON'T RAIN ON MY PARADE)
*(REPLACE RAIN WITH SNOW)
*P
(DON'T SNOW ON MY PARADE)
*OK
(PHRASE)

←phrase
(DON'T SNOW ON MY PARADE)
```

3. If the value of NAME is a list, EDITV evaluates the list. It invokes EDITE (see below) on the result:

```
←(SETQ attributes
      (LIST
            (LIST 'boy (LIST 'male 'young))
            (LIST 'girl (LIST 'female 'young))))
((boy (male young))(girl (female young)))

←(EDITV (CADR (ASSOC 'boy attributes)))
EDIT
*P
(MALE YOUNG)
```

4. Otherwise, EDITV attempts to correct the spelling of the variable name if DWIMFLG is T (see Section 22.2.1). To do so, it uses the list USERWORDS (see Section 22.7.2). EDITV repeats from the first case, if successful.

   If the variable has the value NIL, no spelling correction will be attempted because the variable has a value. However, an error results because the value is not a list and, therefore, not editable.

   EDITV returns the name of the variable whose value it has edited.

19.1.3 Property List Editing

You may edit the property list of an atom by executing **EDITP**, which takes the form

Function:	EDITP
# Arguments:	1-N
Arguments:	1) an atom, ATM 2-N) editor commands, COMMAND[1] ... COMMAND[N]
Value:	The name of the atom.

EDITP is an NLAMBDA, nospread function.

If the atom does not have a property list, EDITP attempts spelling correction using USERWORDS (see Section 22.7.2). Note that here the presumption is that you really want to edit a property list and that it's the atom name that is probably wrong.

EDITP calls EDITE with the property list of the atom, the atom name, and TYPE equal to PROPLIST. When EDITE returns, EDITP establishes the value of the atom's property list via SETPROPLIST as the list returned by EDITE.

EDITP returns the name of the atom whose property list it has edited.

A Definition for EDITP

We might define EDITP as follows:

```
(DEFINEQ
  (EDITP
    (NLAMBDA (editp-arguments)
    (*
        Make EDITP-ARGUMENTS a list so it can be
        dissected. Note that its value may be a single
        atom.
    )
    (AND
       editp-arguments
       (NLISTP editp-arguments)
       (SETQ editp-arguments (LIST editp-arguments)))
    (PROG (name property-list)
        (SETQ name (CAR editp-arguments)
        (COND
            ((NOT (ATOM name))
                (*
                    If the argument is not an atom,
                    just return.
                )
                (RETURN NIL)))
        (SETQ property-list (GETPROPLIST name))
        (*
            Now, if the property list is NIL, assume
            that the user misspelled the atom's name.
            Attempt to correct it if DWIM is enabled.
        )
        (AND DWIMFLG
            (NLISTP property-list)
            (SETQ name
                (OR
                    (MISSPELLED? name NIL USERWORDS)
                    name))
            (SETQ property-list (GETPROPLIST name)))
        (*
            Now, we probably have an atom with a
            property list. If we don't, because the
            atom really doesn't have one, then EDITE
            returns NIL which is just put back in
            place.
        )
        (SETPROPLIST name
```

```
                (EDITE property-list
                       (CDR editp-arguments)
                       name
                       'PROPLIST))
        (RETURN name))
))
```

19.1.4 Editing an Expression

EDITE is the general-purpose expression editor. EDITE has the following generic format:

```
Function:       EDITE

# Arguments:    5

Arguments:      1) an expression, EXPRESSION
                2) an optional command list, COMMANDS
                3) an atom, ATM
                4) an argument type, TYPE
                5) a function invoked if the expression is
                   modified, IFCHANGEDFN

Value:          The expression (possibly modified).
```

EDITE invokes EDITL, the INTERLISP Editor, with EXPRESSION as its argument. It generates an error if EXPRESSION is not a list. The optional command list is a list of zero or more commands that will be applied to the expression.

ATM is the name of the object with which the expression is associated. TYPE gives the nature of the association; it is optional.

IFCHANGEDFN is a function that is applied to the expression if it was changed. The format for invoking IFCHANGEDFN is

```
(⟨IFCHANGEDFN⟩ ⟨atom⟩ ⟨expression⟩ ⟨type⟩ ⟨flag⟩)
```

A Definition for EDITE

We might define EDITE as follows:

```
(DEFINEQ
    (EDITE (expression commands name type ifchangedfn)
        (PROG (an-expression)
            (COND
                ((NLISTP expression)
                    (*
                        If EXPRESSION is not a
                        list, we cannot edit it.
                    )
```

```
                    (PRIN1 expression)
                    (PRIN1 " NOT EDITABLE")
                    (TERPRI)
                    (RETURN expression)))
               (SETQ an-expression
                     (LAST
                         (EDITL (LIST expression)
                                commands
                                name)))
               (COND
                    ((NOT (EQUAL expression an-
                           expression))
                         (*
                              If the expression was
                              really modified, then apply
                              IFCHANGEDFN to it.
                         )
                         (RETURN
                              (AND ifchangedfn
                                   (APPLY* ifchangedfn
                                           name
                                           an-expression
                                           type
                                           flag)))))
               (HELP expression
                     "NOT A LIST")
               (RETURN NIL))
))
```

19.2 EDITL: THE INTERLISP EDITOR

EDITL is the INTERLISP Editor. The generic format for invoking EDITL is

Function:	EDITL
# Arguments:	5
Arguments:	1) a list to be edited, LST 2) an optional list of commands, COMMANDS 3) an atom, ATM 4) a message, MESSAGE 5) an atom whose value reflects the edit changes, EDITCHANGES
Value:	The value of the first argument at the time EDITL is exited.

EDITL edits the list that is its first argument as follows:

1. If COMMANDS is non-NIL, EDITL applies the list of commands to its first argument until an exit condition is encountered.
2. If COMMANDS is NIL, EDITL prints "EDIT" and a prompt character on the terminal and waits for the user to enter commands to edit the first argument.

In interactive editing, an exit will occur only when one of the commands—OK, STOP, or SAVE—is recognized.

All input to the Editor is affected by the current values in EDITRDTBL.

MESSAGE is typed on the terminal if it is non-NIL. It will not be displayed if COMMANDS is non-NIL, i.e., it serves as an alternative message to the user in lieu of "EDIT".

ATM is optional. It refers to the object that is being edited. Its type depends on the calling function. Note that the SAVE command uses the property list of the atom to save the current state of the edit. Of course, nothing will be saved if the atom name is NIL.

EDITCHANGES is used to store a list of changes to the first argument.

An alternative form, **EDITL0**, operates like EDITL except it does not rebind or initialize the Editor's state variables.

19.3 EDITOR FUNCTIONS

The Editor relies on a number of "workhorse" functions to perform its operations. These functions may also be called from your programs to perform the basic editing operations.

19.3.1 Finding a Pattern

EDIT4E is the basic pattern matching routine of the Editor. It takes the form

```
Function:       EDIT4E
# Arguments:    2
Arguments:      1) a pattern, PATTERN
                2) an expression, EXPRESSION
Value:          T, if PATTERN matches EXPRESSION.
```

EDIT4E matches PATTERN according to the definition provided in Section 19.4.4.

Before each search operation, the Editor scans the entire pattern for atoms or strings containing the character sequence $s (or ⟨ESC⟩s). Each atom or string is replaced by a list of the form ($ ...). A similar approach is taken for

atoms or strings ending in $$s (or ⟨ESC⟩⟨ESC⟩s). These patterns are detected by taking the CAR of PATTERN. If you want to detect these from your program, you must convert your expressions to this form.

EDITFPAT makes a copy of its argument, PATTERN, with all atoms or strings containing $s (or ⟨ESC⟩s) converted to the form expected by EDIT4E.

Implementing the Find Command

You may search for a pattern using the Editor's Find command using **EDITFINDP**, which takes the form

```
Function:       EDITFINDP

# Arguments:    3

Arguments:      1) an expression, EXPRESSION
                2) a pattern, PATTERN
                3) a flag, FLAG

Value:          T, if the PATTERN is found in EXPRESSION
                because F PATTERN would succeed; NIL,
                otherwise.
```

EDITFINDP executes the Editor's Find command as a pure predicate. If FLAG is NIL, EDITFPAT is invoked to convert PATTERN to the form expected by EDIT4E. Thus, you may call EDITFPAT once, and EDITFINDP several times if you are looking for several different expressions.

19.3.2 Substituting in an Expression

The Editor's Replace command may be executed directly using **ESUBST**, which takes the form

```
Function:       ESUBST

# Arguments:    5

Arguments:      1) a new expression, NEW
                2) an old expression, OLD
                3) an expression, EXPRESSION
                4) an error flag, ERRORFLAG
                5) a character flag, CHARFLAG

Value:          The modified EXPRESSION.
```

ESUBST performs the Editor's Replace command that is equivalent to (R ⟨old⟩ ⟨new⟩). EXPRESSION is assumed to be the current expression. If OLD is not found in EXPRESSION, an error is generated. If ERRORFLAG is T, an error message of the form "OLD ?" will be printed.

If CHARFLAG is T and no forms like $s (or ⟨ESC⟩s) are specified in NEW or OLD, ESUBST operates like (RC ⟨old⟩ ⟨new⟩).

ESUBST is always undoable.

19.3.3 Changing Names

CHANGENAME is used to change the names of objects in a function. It is primarily used by the Break and Advising Packages to change the name of a function within another function. It takes the form

```
Function:       CHANGENAME
# Arguments:    3
Arguments:      1) a function, FN
                2) the old name, FROM
                3) the new name, TO
Value:          Either FN or NIL.
```

CHANGENAME replaces all occurrences of FROM by TO in the specified function. CHANGENAME succeeds whether FN has a symbolic form, e.g., it is an EXPR, or it is compiled. CHANGENAME returns FN if at least one instance of FROM is found in the definition.

19.3.4 Searching Files

EDITCALLERS provides a mechanism for rapidly searching a single file or set of files, whether they are loaded or noticed or not. It takes the form

```
Function:       EDITCALLERS
# Arguments:    3
Arguments:      1) a list of atoms, ATOMS
                2) a list of files, FILES
                3) a list of commands, COMS
Value:          The list of files.
```

EDITCALLERS uses FFILEPOS to search each file in FILES for occurrences of the atoms in ATOMS. For each occurrence that is found, EDITE is invoked to apply the commands in COMS to the detected atoms. COMS may be NIL, whence (EXAM . ⟨atoms⟩) is used. Both ATOMS and FILES may be single atoms. If FILES is NIL, FILELST is used as the default list of files.

EDITCALLERS prints the name of each file that it searches. When it finds an occurrence of an atom in ATOMS, it prints either the name of the function containing the atom or the byte position of the atom in the file. If a file map is

available, EDITCALLERS uses it to speed up the search. EDITCALLERS always calls the Editor.

An alternative form, **FINDCALLERS**, which takes the arguments ATOMS and FILES, merely performs the search, but never calls the Editor.

19.3.5 Tracing Editor Macros

EDITRACEFN is used to help you debug edit macros. It takes the form

```
Function:       EDITRACEFN
# Arguments:    1
Arguments:      1) a command, COMMAND
Value:          The value of the trace or break.
```

EDITRACEFN is initially NIL. If you set it to T, then EDITRACEFN will be invoked whenever a command not typed by you is about to be executed. The command may be an edit macro or a valid editor command. The command is passed to EDITRACEFN as its argument. You may define EDITRACEFN to handle the command as you see fit.

If EDITRACEFN is set to TRACE, the name of the command and the current expression are printed. If EDITRACEFN is BREAK, the same information is printed and the Editor enters the Break Package. In the Break Package, you may examine the state of the Editor.

19.4 EDITOR CONCEPTS

Editor commands can be divided into a basic set that are easy to use and an advanced set that provide substantially more power and flexibility to the experienced user. To understand how the Editor works, it is necessary to understand the model for editing that the editor works with. This section describes these concepts.

19.4.1 The Concept of Currency

The INTERLISP editor works on list structures. As you know, lists are bounded by matched pairs of parentheses. Within each pair of matching parentheses, multiple lists and atoms—the *elements* of a list—may exist. The editor works within a matching pair of parentheses on one of the elements.

Consider the following definition for a function:

```
(LAMBDA (x)
    (COND
        ((ZEROP a) x)
```

```
        (T
            (MINUS x))))
```

The top-level elements of this list structure are LAMBDA, (x), and (COND & &). The latter two elements are sublists having further structure. The &'s mean that there is yet greater complexity in a sublist.

Using normal list processing terminology, sublists always have a level that is one greater than the level in which they are contained. Thus, the level of x in (x) is 2 where we count the top level as 1. Correspondingly, the level of MINUS is 4 since it is located in the sublist of a sublist of a sublist.

The editor works, usually, on one element of a list at a time. In order to access the elements of a sublist, we must descend into the sublist. To descend in to the sublist given by (COND & &), we must specify its element number to the editor (see Section 19.5.2). When we do so, the elements of that sublist become visible to us and the editor can manipulate them.

In order to keep track of where it is in a list structure, the editor maintains an *edit chain*. This is merely a list structure on which is pushed the current expression when the editor descends into one of its elements.

19.4.2 The Print Level

The Editor adheres to the print-level specification established by the system variable PRINTLEVEL. The Editor assumes an initial print level of (2 . 20).

19.4.3 Multiple Commands per Line

You may combine multiple commands per type-in line. The Editor will execute each command in turn. However, an error in any command will cause the termination of the command sequence.

19.4.4 Pattern Specifications for Searching

All of the Editor commands that search use the same pattern matching routine. A *pattern* matches with an expression in the current expression if its satisfies any of the following conditions:

1. If the ⟨pattern⟩ is EQ to the expression.
2. If the ⟨pattern⟩ is the atom &.
3. If the ⟨pattern⟩ is a number and EQP to the expression.
4. If the ⟨pattern⟩ is a string and

   (STREQUAL ⟨pattern⟩ ⟨expression⟩)

   is true.

5. If (CAR ⟨pattern⟩ is the atom *ANY* and (CDR ⟨pattern⟩) is itself a list of patterns, one of which matches the expression.
6. If ⟨pattern⟩ is a literal atom or string containing one or more $⟨s⟩ (e.g., ⟨ESC⟩⟨s⟩) where each $ can match an indefinite (including 0) number of contiguous characters in the atom or string.
7. If ⟨pattern⟩ is a literal atom or string ending in two ⟨ESC⟩s, the ⟨pattern⟩ matches with an atom or string that is close to ⟨pattern⟩, in the same sense that the spelling corrector uses.
8. If (CAR ⟨pattern⟩) is the atom -- (e.g., two dashes), ⟨pattern⟩ matches the expression if (CDR ⟨pattern⟩) matches with some tail of the expression. If (CDR ⟨pattern⟩) is NIL, then ⟨pattern⟩ matches any tail of the expression.
9. If (CDR ⟨pattern⟩) is the atom == (e.g., two equal signs), the ⟨pattern⟩ matches the expression if and only if (CDR ⟨pattern⟩) is EQ to the expression.
10. If (CADR ⟨pattern⟩ is the atom .. (e.g., two periods), ⟨pattern⟩ matches the expression if (CAR ⟨pattern⟩) matches (CAR ⟨expression⟩), and (CDDR ⟨pattern⟩) is contained within the expression.
11. Otherwise, if the expression is a list, ⟨pattern⟩ matches the expression if (CAR ⟨pattern⟩) matches (CAR ⟨expression⟩) and (CDR ⟨pattern⟩) matches (CDR ⟨expression⟩).

The significance of these rules is described in more detail in the IRM.

19.5 BASIC EDITOR COMMANDS

This section will discuss the basic editor commands that will allow an unsophisticated user to "get started" with the editor.

19.5.1 Printing the Current Expression

You may print the current expression by executing the command **P**. If the current expression is the top level, then we would see

```
*P
(LAMBDA (a 1) (COND & & &))
```

where the & indicates that a sublist resides here.

The Editor assumes an initial print level of (2 . 20), which means that sublists of sublists will be printed [i.e., (a 1)] and the tails of long lists will be abbreviated as --.

If you wish to see the entire expression to be edited, you may issue the command **?**, which forces the Editor to assume a print level of 1000. Thus, we have

```
*?
(LAMBDA (a 1) (COND ((NULL 1) NIL) ((EQUAL (CAR 1) a) T) (T
(MEMBER a (CDR 1)))))
```

Note that the Editor does not observe any specific demarcations when it prints an expression in this manner.

To prettyprint an expression in a more legible format, you may use the command **PP**:

```
*PP
(LAMBDA (a 1)
     (COND
          ((NULL 1) NIL)
          ((EQUAL (CAR 1) a) T)
          (T (MEMBER a (CDR 1)))))
```

19.5.2 Descending a Level

A positive integer directs the Editor to descend into the corresponding element of the current expression. If the element is an atom, the Editor prints an error indication consisting of the level number followed by a ?.

```
*2
(a 1)          "the second element"
*1
1 ?            "cannot descend into an atom"
```

Note that the current expression is never changed when a command fails, so

```
*P
(a 1)
```

A negative integer forces the Editor to count backward from the end of the current expression to select the element into which to descend.

```
*-2
(a 1)                "again, the second element"
*-3
-3 ?                 "cannot exceed number of elements"
```

Note that whenever we descend into an element, the current expression is always added to the edit chain.

19.5.3 Ascending the Edit Chain

To ascend the edit chain, you may issue the command **0**. 0 operates by removing the last link of the edit chain and making it the current expression.

```
*P
(NULL 1)              "assume we are at the 4th level"
*0
(& NIL)               "to the 3rd level"
*0
(COND & & &)          "to the 2nd level"
```

Note that 0 ascends one level each time that it is executed. If we are already at the top level of the list structure, 0 will cause an error because the Editor cannot remove any more links from the edit chain, i.e., the edit chain must always be anchored by the top level of the list structure.

In a particularly complex list structure, you may forget at what level of the list you currently reside. To "pop" up to the top level, you may execute the ↑ (up-arrow) command to force the editor to ascend to the top-most level. For example,

```
*P
(CDR 1)
*↑ (up-arrow)
(LAMBDA (a 1) (COND & & &))
```

If you are already at the top-most level of the list structure, the (up-arrow) command has no effect.

19.5.4 Modifying the List Structure

The structure modification commands always work on the current structure. The general form of the structure modification command is (# ⟨**expression-list**⟩) where the ⟨expression-list⟩ may be empty. The # may be either positive or negative and determines the element in the current expression that is to be modified.

To delete an element from the current expression, you may issue the command (#) where # is the index of the corresponding element to be deleted.

```
*P
(COND (& NIL) (& T) (T &))
*(2)
*P
(COND (& T) (T &))
```

where the element (& NIL) has been deleted. Note that the new structure is not automatically printed after modification.

Structure modification performed by the Editor is destructive, i.e., it physically modifies the list structure using the functions RPLACA and RPLACD.

To replace an element with one or more new elements, you may issue the command (# **e[1] ... e[N]**):

```
*P
(& NIL)
*(2 T)
*P
(& T)              "NIL is replaced by T"
```

To insert one or more new elements before a given element, you may issue the command **(-# e[1] ... e[N])**. The sign of the number indicates that an insertion is to take place.

```
*P
(COND & & &)
*(-2 ((NULL a) NIL))
*P
(COND & & & &)
```

where a complex expression has been inserted before the second element.

Note that all three commands execute with respect to the Nth element from the beginning of the current expression. There is no way to specify insertion or deletion from the end of the current expression. Nor can we specify insertion after a given element. This is rather tedious and may cause you some difficulty at first.

19.5.5 Adding Elements to the End of the Current Expression

Because we cannot insert after an element, we cannot put something at the end of the list using structure modification commands described in Section 19.5.4. To add elements to the end of the current expression, you may use the **N** command. The N command NCONCs the elements provided to the end of the current expression.

```
*P
(LAMBDA (a l) (COND & & &))
*3 P
(COND (& NIL) (& t) (T &))
*(4)
*(N (T (EQUAL a (CADR l))))
*4 P
(T (EQUAL a (CADR l)))
```

Note that the N command performs an RPLACD on the last CDR of the current expression in order to patch in the new expression as the last element of the current expression.

19.5.6 Finding an Element

One way to determine where to modify a structure is to perform a sequence of recursive descents into the structure. For small structures, this may be quite efficient and easy, particularly when the entire structure is composed of no more than a few lines of code. More than this requires substantial effort on your part to keep track of where you are.

The Find command allows you to *search* a structure for a pattern. The format of the Find command is

```
*F ⟨pattern⟩
```

where ⟨pattern⟩ is the thing to be looked for in the list structure.

When the Find command is executed, the Editor begins searching at the current level for the pattern. If it does not find it at the current level, it ascends one level (via an implied UP) and renews the search. However, the search proceeds only in the forward direction. Thus, when the Editor ascends a level, it searches only those elements *following* the element at which the search began.

If the search succeeds, the Editor sets the current expression to the structure element where the pattern is found. If the search fails, the current expression is not changed and an error message is displayed.

Consider the following example:

```
*P
 (LAMBDA (A L) (COND & & &))
*3 P
 (COND (& NIL) (& T) (T &))
*2 P
 ((NULL L) NIL)
*F COND
 COND ?              "look for a COND"
```

The search failed to find a COND either in the current expression or any expressions following it in the structure. Note that a COND appears earlier in the structure but the Find command does not backtrack to locate it. To search from the previous level, we might have issued the command

```
*0 F COND P
```

whence the result would be

```
(COND (& NIL) (& T) (T &))
```

19.5.7 Replacing an Element

In Section 19.5.4, we saw that we could replace an element of a structure by descending to that element and specifying a new element in its entirety. If we want to replace one atom of an element, this may become rather tedious.

The **R**eplace command allows us to locate an atom or subexpression of the current expression and substitute a new atom or expression for it. Consider the following example:

```
*P
 (LAMBDA (A L) (COND & & &))
*3 P
 (COND (& NIL) (& T) (T &))
*(R EQ EQUAL)
*PP
 (LAMBDA (A L)
     (COND
          ((NULL L) NIL)
          ((EQUAL (CAR L) A) T)
          (T (MEMBER A (CDR L)))))
```

The Replace command *replaces* all occurrences of the first argument by the second argument in the current expression. In this case, there was only one occurrence for which the substitution was made.

The Replace command requires that there be at least one occurrence where substitution takes place; otherwise, an error occurs.

19.5.8 Exiting the Editor

When you complete the editing of a structure, you may exit the editor by issuing the **OK**ay command. OK saves the edited result in the proper place according to how it was invoked.

19.6 AN EDITOR COMMAND ENCYCLOPEDIA

The INTERLISP Editor provides a rich, powerful, flexible set of commands for manipulating list structures. Beyond the brief introduction given in Section 19.5, it is difficult to instruct a user in the various Editor commands. We have found the best method is to wade in and try the various commands. You will feel comfortable with some but less so with others. Eventually, you will build a repertoire of commands that you will use frequently. When you become "stuck," be assured the Editor probably has a command for what you want to do, but you will have to look it up.

In this spirit, this section presents an encyclopedia of Editor commands. The following table depicts the correspondence between the command, its function, and the section in which it appears.

Command	Function	Section
A	After	19.6.1
B	Before	19.6.2
BELOW	Below	19.6.3
BF	Backwards Find	19.6.4
BI	Both In	19.6.5
BIND	Binding Variables	19.6.6
BK	Backup	19.6.7
BO	Both Out	19.6.8
CAP	Capitalize	19.6.9
CHANGE	(See REPLACE)	
CL	Clispify	19.6.10
COMS	Commands	19.6.11
DELETE	Deletion	19.6.12
DW	Dwimify	19.6.13
E	Evaluate	19.6.14
EMBED	EMBEDding	19.6.15
EVAL	EVALuation	19.6.16
EXAM	EXAMine	19.6.17
EXTRACT	(See XTR)	
F	Finding an expression	19.6.18
GETD	GETting a Definition	19.6.19
GETVAL	GETting a VALue	19.6.20
GET*	GETting a comment	19.6.21
GO	GOing to a PROG label	19.6.22
IF	Conditional Editing	19.6.23
INSERT	Inserting into an expression	19.6.24
JOINC	Joining Conditional expressions	19.6.25
LC	Lower Case	19.6.26
LI	Left Insert	19.6.27
LOWER	Lower Case	19.6.28
LP	LooPing	19.6.29
M	Macro definition	19.6.30
MAKE	MAKE a parameter have a value	19.6.31
MAKEFN	MAKE a FunctioN	19.6.32
MARK	MARKing and Restoring the edit chain	19.6.33
MBD	(see EMBED)	
MOVE	MOVE an expression	19.6.34
N	Adding to the eNd of an expression	19.6.35
NEGATE	NEGATEing an expression	19.6.36

```
3RD                 (see LC)
:                   (see DELETE)
?                   (see P)
\                   (see MARK)
←┤                  (see UP)
←                   (see MARK)
```

Many of the Editor commands are treated as macros. Their definitions are maintained in the system variable EDITMACROS. You may discover these definitions by executing

```
←(PRINTDEF EDITMACROS)
```

We shall show the definition for each macro that appears on EDITMACROS.

19.6.1 Inserting After the Current Expression

The After command allows you to insert one or more S-expressions after the current S-expression. Its format is

```
*(A e[1] ... e[n])
```

where the e[i] are S-expressions. Consider the following examples:

```
←(EDITE x)
edit
*PP
 (COND
     ((EQ A (CAR L))
          (CDR L)))
*(A ((EQ (CAR A) (CAR L)) (CADR L)) (T L))
can't -- at top
```

because we are trying to add an S-expression after an entire current S-expression. However, we really want to descend into the S-expression before editing:

```
*2 P
((EQ A &) (CDR L))
*(A ((EQ (CAR A) (CAR L)) (CADR L)) (T L))
*PP
(((EQ A (CAR L))
      (CDR L))
 ((EQ (CAR A) (CAR L))
      (CADR L))
```

```
 (T L))
*0 P
(COND (& &) (& &) (T L))
```

19.6.2 Inserting Before the Current Expression

The **B**efore command allows you to insert one or more S-expressions before the current expression. Its format is

```
*(B e[1] ... e[n])
```

where the e[i] are S-expressions. Consider the following examples:

```
←(EDITE x)
edit
*PP
(COND
     ((EQ A (CAR L))
          (CDR L)))
*(B ((AND (LISTP A) (EQ (CAR A) (CAR L))) (CADR L)))
can't -- at top
```

because we are trying to add an S-expression before an entire S-expression. We must descend into the S-expression as follows:

```
*2 P
((EQ (CAR A) (CAR L))
*(B ((AND (LISTP A) (EQ (CAR A) (CAR L))) (CADR L)))
*P
(((AND (LISTP A)
       (EQ (CAR A)
           (CAR L)))
       (CADR L))
 ((EQ A (CAR L))
        (CDR L)))
```

19.6.3 Locating a Pattern

The **BELOW** command is used to locate current expression after a given pattern in a structure. Its format is

```
*(BELOW ⟨pattern⟩ ⟨n⟩)
```

The Editor ascends the edit chain looking for a link specified by ⟨pattern⟩. Ascension consists of performing 0s (see Section 19.6.60) until a link is found

that matches ⟨pattern⟩. Then, the editor descends ⟨n⟩ links below that expression.

Consider the following example taken from the IRM:

```
*PP
     (PROG NIL
          (COND
               ((NULL (SETQ L (CDR L)))
                    (COND
                         (FLG (RETURN L))))
               ((NULL (CDR (MEMBER (CAR L) (CADR L)))))))
*F CADR
*(BELOW COND)
*P
((NULL (CDR (MEMBER (CAR L)(CADR L)))))
```

F CADR finds the S-expression (CADR L). Then, (BELOW COND) ascends the edit chain four links:

```
(MEMBER & &)
(CDR &)
(NULL &)
(NULL &)              "contains a COND"
```

which is equivalent to

```
*0 0 0 0
```

The BELOW command forces the result shown above. Since ⟨n⟩ is NIL, 1 is assumed.

BELOW locates a substructure by specifying an S-expression that it contains.

19.6.4 Searching Backwards

The **B**ackwards **F**ind command searches backwards from the current expression to locate an S-expression matching the pattern. Its format is

```
*BF ⟨pattern⟩
```

Consider the following examples:

```
*P
     (PROG NIL
          (SETQ X
```

```
                    (SETQ Y (LIST Z)))
          (COND
               ((SETQ W (CAR X))
                    (CADR Z))))
*F LIST
*BF SETQ
*P
(SETQ Y (LIST Z))
```

When BF ascends the edit chain to the previous link, it always begins its search at the end of the link. That is, it searches the list in reverse order. So, F LIST finds (LIST Z). Then, BF SETQ backs up to (SETQ Y (LIST Z)) which contains a SETQ and causes the search to terminate.

BF continues ascending until the top of the edit chain is reached, whence it terminates with ?.

```
*F CADR
*BF RETURN
 ?
```

If the current expression is the top-level expression, the editor searches the entire expression in reverse order.

```
*↑                              "up arrow"
*BF CAR
*P
(CAR X)
```

An alternative form, (BF ⟨pattern⟩ T), causes the current expression to be included in the search.

```
*F COND
*(BF SETQ T)
*P
(SETQ W (CAR X))
```

because the search commences at the end of (COND &) which is the current expression.

BF is constrained by the value of MAXLEVEL and UPFINDFLG.

19.6.5 Inserting Balanced Parentheses

The **Both In** command allows you to insert balanced parentheses around a sequence of elements in the current expression. Its format is

```
*(BI ⟨n⟩ ⟨m⟩)
```

where ⟨n⟩ and ⟨m⟩ are the numeric indices of expressions within the current expression.

Consider the following example:

```
←(EDITF is.zero)
edit
*PP
(LAMBDA (NUM)
     (COND
          (ZEROP NUM 1)
          (T NUM)))
*F ZEROP
*P
(ZEROP NUM 1)
*(BI 1 2)
*P
((ZEROP NUM) 1)
```

BI inserts a left parenthesis *before* the ⟨n⟩th element in the current expression and a matching right parenthesis *after* the ⟨m⟩th element. ⟨m⟩ must be greater than ⟨n⟩ and the ⟨m⟩th element must be contained within the current expression. Otherwise, BI generates an error.

```
*(BI 1 4)
4 ?
*(BI 2 1)
(BI 2 1) ?
```

An alternative form, (BI ⟨n⟩), is equivalent to (BI ⟨n⟩ ⟨n⟩). That is, it makes a list of the ⟨n⟩th element.

19.6.6 Binding Macro Variables

The **BIND** command binds the dummy variables #1, #2, and #3, which are initialized to NIL, and then executes a sequence of edit commands. Its format is

```
*(BIND ⟨commands⟩)
```

The IRM notes that you may define the SW command as

```
(M
     (SW (N M)
          (BIND (NTH N)
                (S #1 1)
                MARK 0
                (NTH M)
```

```
                (S #2 1)
                (I 1 #1) ←← (I 1 #2))))
```

The commands (S #1 1) and (S #2 1) set the macro variables #1 and #2 appropriately.

The bindings for macro variables #1, #2, and #3 are effective only for the sequence of edit commands executed by BIND.

19.6.7 Backing Up in the Current Expression

A current expression may be embedded in a list. **BacKup** allows you to move to the previous expression in the edit chain,

```
*PP
     (COND
          ((NULL X)
               (RETURN Y)))
*F RETURN
*P
     (RETURN Y)
*BK P
     (NULL X)
```

where the list structure consists of ((NULL X) (RETURN Y)).

If the current expression is at the beginning of a list, an error will result. That is, BK does not ascend the edit chain.

```
*PP
     (COND
          ((NULL X)
               (RETURN Y)))
*F NULL
*P
     (NULL X)
*BK
BK ?
```

An alternative form, (BK ⟨n⟩), allows you to specify the number of S-expressions, ⟨n⟩ => 1, to back up within the current list structure. If ⟨n⟩ would force BK to run off the front of the list, an error results and the edit chain is not changed.

```
*PP
     (tyler polk johnson hayes taft arthur)
*F taft
*(BK 5)
(BK 5) ?
```

A Definition for BK

We might define a macro which backs up to a previous expression and prints it as follows:

```
(BKP NIL
     (ORR (BK)
          (!0)
          ((E
               (PROGN
                    (SETQQ COM BK)
                    (ERROR!)))))
     P)
```

19.6.8 Deleting Balanced Parentheses

The **B**oth **O**ut command allows you to remove balanced parentheses from an element in an expression. Its format is

```
*(BO ⟨n⟩)
```

where ⟨n⟩ identifies the Nth element in the expression.

Consider the following example:

```
*P
((A B) C (D E F) G (H I))
*(BO 3)
*P
((A B) C D E F G (H I))
```

If the Nth element is not a list or there are not N elements in the expression, an error is generated.

```
*P
((A B) C (D E F) G (H I))
*(BO 4)
(BO 4) ?
*(BO 6)
(BO 6) ?
```

19.6.9 Capitalization

The **CAP**italization command capitalizes an atom or a string. It RAISEs the characters in the element and then LOWERs all but the first character. If the atom or string is already capitalized, nothing happens.

```
*P
IGNORECHARACTERCODES
*CAP
*P
Ignorecharactercodes
```

CAP's macro definition appears as

```
(CAP NIL
     RAISE (I 1 (L-CASE (##1) T)))
```

19.6.10 CLISPIFYing Expressions

The **CLISPIFY** command translates the current expression to CLISP format (see Chapter 23).

```
*P
(COND
     ((NULL x)
          (PROG (value)
           loop
                (COND
                    ((EQ (SETQ x (READ) 'STOP)
                         (RETURN value)))
                (PRINT (EVAL x))
                (GO loop))))
*CL
*P
(if (NULL x)
    then while x←(READ) = 'STOP
         do (PRINT (EVAL x)))
```

A Definition for CL

CL has the following macro definition:

```
(CL NIL
    (BIND
          (IF (NULL (CDR L))
              (IF (MEMB (##1) LAMBDASPLST)
                  ((MARK #3) 3 UP)
                  ((E
                      (PROGN
                         (SETQQ COM CL)
                         (PRINT 'can't) T T)
                         (ERROR!))))
```

```
     NIL)
(IF (TAILP (SETQ #1 (##))
           (##!0
              (E
                 (SETQ #2 L) T)))
     ((I : (CLISPIFY #1 #2))
      (LO 1))
     ((COMS
          (CONS ': (CLISPIFY #1 #2))
          (AND (LISTP (##1) 1))
(IF #3 (( #3)) NIL)))
```

19.6.11 Command Execution

COMS takes one or more S-expressions as its arguments. It evaluates each S-expression in turn and then executes it as an Editor command. Its format is

```
*(COMS expression[1] ... expression[N])
```

where each EXPRESSION[i] is an S-expression which, when evaluated, yields an Editor command.

An alternative form, **COMSQ**, takes as its arguments one or more S-expressions which it executes as Editor commands (e.g., its arguments are Editor commands). Its format is

```
*(COMSQ command[1] ... command[N])
```

where each COMMAND[i] is an Editor command.

COMS and COMSQ may be used in conjunction to execute commands that are attached to an INTERLISP object that is being edited, perhaps as entries on its property list. Different properties may describe how to edit the data structure.

19.6.12 Deleting Expressions

The **DELETE** command deletes the current expression on the edit chain:

```
*P
(COND
     (A (PRINT B))
     (T (HELP)))
*F HELP
*DELETE
*P
(T)
```

DELETE tries to delete the current expression by performing an UP and then (1). This works in all cases except where the current expression contains exactly one element, because you cannot delete something and leave an empty list. Then, DELETE performs, BK, UP, (2).

When the next higher expression contains only one element, BK will not succeed. Then, DELETE performs (: NIL). That is, it replaces the higher expression by NIL.

DELETE might be defined as an edit macro as follows:

```
(DELETE NIL
     (ORR (UP (1))
          (BK UP (2))
          (: NIL)))
```

An alternative form, **D**, combines the actions of DELETE and P by deleting the current expression and printing the new current expression. It might be defined as

```
(D NIL (:) 1 P)
```

Another form, **:**, also deletes the current expression.

19.6.13 DWIMIFYing Expressions

The **DW**IMIFY command allows you to DWIMIFY the current expression (see Section 22.5). The process of DWIMIFYing is described in detail in Chapter 22.

A macro definition for the DW command is

```
(DW NIL
    (BIND
          (E
               (PROGN
                    (SETQ #1 (##))
                    (AND
                         (CDR L)
                         (##!0 (E (SETQ #2 L) T)))
                    (AND
                         (SETQ #3
                              (DWIMIFY #1 T (OR #2
                              'NIL)))
                         EDITCHANGES
                         (RPLACA (CDR EDITCHANGES) T)))
          T)
          (IF (NLISTP #1)
              ((I : #3)
```

```
        (IF (LISTP #3)
            (1)
            NIL))
    NIL)))
```

You may disable DWIMIFYing in the Editor using the following command:

```
(!DW NIL
     (RESETVAR CLISPRETRANFLG T DW))
```

19.6.14 Evaluating Input

From the Editor, you may evaluate any S-expression. To do so, you must enter the Evaluate command followed by the expression to be evaluated. Note that E only works when typed in at your terminal. It takes the form

```
*E ⟨expression⟩
```

where the expression is evaluated. For example, while editing, you may want to evaluate part of an expression, see what it produces, and force a break. Using E, you enter

```
*(E (LOAD 'COMPLEX))
⟨KAISLER⟩COMPLEX..2
FILE CREATED 30-Aug-84 20:44:36
COMPLEXCOMS
```

An alternative form is (E ⟨expression⟩). ⟨expression⟩ is evaluated and then E performs an EVAL (see below) on its result. The value is printed at your terminal.

```
*(E (SETQ X '(IPLUS (SETQ X 1) (SETQ Y 4))))
(SETQ X '(IPLUS (SETQ X 1)(SETQ Y 4)))
```

If you do not want the result to be printed at your terminal, you may use the form

```
*(E (SETQ X '(IPLUS (SETQ X 1) (SETQ Y 4))) T)
```

which suppresses printing of the result of the expression, except during type-in mode.

I have noticed that this command is particularly useful when you are editing broken functions because you may inspect the values of variables or subexpressions of the function from within the Break Package.

19.6.15 Embedding

Embedding replaces the current expression with one containing it as a subexpression. There are two forms to this command:

```
*(EMBED @ IN ⟨expression⟩)
```

where @ is some S-expression (possibly, an atom) which is to be inserted into the S-expression given by ⟨expression⟩.

```
←(EDITF ROUNDTO)
edit
*PP
(LAMBDA (X Y)
     (TIMES (ROUNDED (QUOTIENT X Y)
                     Y)))
*(EMBED QUOTIENT IN PRINT)
*PP
(LAMBDA (X Y)
     (TIMES (ROUNDED (PRINT (QUOTIENT X Y)) Y)))
```

EMBED locates the S-expression identified by @ via (LC . @). It then creates a new S-expression with @ embedded via (MBD . ⟨expression).

MBD is the command that performs embedding. It takes the form:

```
*(MBD ⟨expression1⟩ ... ⟨expressionN⟩)
```

MBD substitutes the current expression for all instances of the atom & in each of the ⟨expression[1]⟩. The current expression is replaced by the result of that substitution via SUBST. MBD sets the edit chain to be the new current expression.

Another alternative, **Q**, quotes the current expression by embedding a QUOTE in front of the expression. It takes the form

```
*Q
```

Q may be defined as an edit macro as follows:

```
(Q NIL
   (MBD QUOTE))
```

Another alternative form, **SURROUND**, performs the same function as EMBED but is easier understood. It takes the form

```
*(SURROUND @ WITH . ⟨expression⟩)
```

19.6.16 Evaluating an Expression

You may evaluate the current expression in the edit chain by issuing the **EVAL**uate command. It takes the form

```
*EVAL
```

You may use EVAL and NX to single-step a program through its symbolic definition. Assume we have the PROG form:

```
*E (SETQ CITIES (LIST 'WICHITA 'ALBANY 'BOISE 'SALEM))
*PP
(PROG (CITY)
LOOP (SETQ CITY (CAR CITIES))
      (PRIN1 CITY)
      (TERPRI)
      (SETQ CITIES (CDR CITIES))
      (AND
          (NULL CITIES)
          (RETURN))
      (GO LOOP))
```

The execution of the PROG would be handled by the following sequence of commands:

```
*F SETQ
*P
(SETQ CITY (CAR CITIES))
*EVAL
WICHITA
*NX
*P
(PRIN1 CITY)
*EVAL
WIHCITAWICHITA
```

A macro definition for EVAL is

```
(EVAL NIL
      (ORR
          (E
               (LISPXEVAL (## (ORR UP1) NIL)) '*)
          ((E 'EVAL?))))
```

19.6.17 Examining an Expression

EXAMine allows you to invoke the Editor on an expression from within the Editor. Its form is

```
*(EXAM . ⟨expression⟩)
```

⟨expression⟩ is a pattern or list of patterns. The Editor will be invoked for every instance of the expression. Assuming we are in the PROG expression mentioned in Section 19.6.16, we have

```
*(EXAM . CITIES)
(SETQ CITIES (CDR CITIES))
tty:
*P
(SETQ CITIES (CDR CITIES))
*OK
(CDR CITIES)
tty:
*OK
(NULL CITIES)
tty:
*OK
done
*
```

where the Editor has stepped us through every instance where CITIES occurs in the remaining S-expressions from the current edit chain. At each instance, we can enter editing commands to modify that particular instance.

We might define EXAM as a macro as follows:

```
(EXAM X
      (F (*ANY* . X) T)
      (BIND (LPQ (MARK #1)
            (ORR (1 !0 P) NIL)
            (MARK #2)
            TTY:
            (MARK #3)
            (IF (EQ (##(\ #3))
                    (##(\ #2)))
                ((\ #1))
                NIL)
            (F (*ANY* . X) N)))
      (E 'done))
```

19.6.18 Finding an S-expression

When editing a function or expression, you must be able to locate the expression within the function to be modified. The Find command and its variations search an expression for a given pattern. If the search is successful, the current expression becomes the pattern sought.

The basic form of the F command is

```
*F ⟨pattern⟩
```

where ⟨pattern⟩ is to be sought in the function or expression that is being edited. If the pattern exists within the current expression, F does not proceed to search further, i.e., F initially executes

```
(MEMBER ⟨pattern⟩ ⟨current-expression⟩)
```

Consider the following examples (after the IRM):

```
*PP
(PROG NIL
      (SETQ ...)
LOOP
      (COND
          ((NULL ...) ...)
          ((MEMBER ...)(GO LOOP1)
          (T (GO LOOP)))
LOOP1 (RETURN))
*1
*F LOOP1
*P
... LOOP1)
```

finds the first LOOP1 which is embedded within the COND because PROG is the current expression. The ... indicates that there is a part of the S-expression preceding the atom that was found. However,

```
*0 P
*F LOOP1
```

finds the label LOOP1 within the PROG.

You can find the next instance of a pattern by using the alternative form

```
*F ⟨pattern⟩ N
*(F ⟨pattern⟩ N)
```

This form does not perform the check against the current expression.

You may determine if there is another instance of the pattern in the function or expression using the alternative form

```
*F ⟨pattern⟩ T
*(F ⟨pattern⟩ T)
```

This form looks for the pattern, but does not change the edit chain when it finds an instance of the pattern. It does not check the current expression.

If there are multiple instances of a pattern in the function or expression, you can find the Nth instance using the alternative form

```
*(F ⟨pattern⟩ ⟨n⟩)
```

where ⟨n⟩ is greater than or equal to 1. This form is equivalent to

```
*(F ⟨pattern⟩ T)
*(F ⟨pattern⟩ N) repeated N-1 times.
```

If the pattern does not match N times, an error is generated and the edit chain remains unchanged.

```
*F LOOP1 4
LOOP1 ?
```

The search algorithm used by F descends into lists to match its pattern. If you want to match only the top-level elements of the current expression, you may use the alternative form (differing from the basic form by enclosure within parentheses)

```
*(F ⟨pattern⟩)
*F ⟨pattern⟩ NIL
```

Consider the following example:

```
*P
(PROG (...)
      (SETQ ...
            (COND
               ((...))
               ((...))))
      (COND
         ((...))
```

```
          (T ...)))
*F COND
```

will find the COND inside the SETQ because that form descends into lists.

```
*(F COND)
```

finds the second COND because it occurs at the top level of the PROG form.

Finding Any Pattern
An alternative form, **ORF**, searches for an expression that matches any one of the patterns given as its arguments. It takes the form

```
*(ORF ⟨pattern1⟩ ... ⟨patternN⟩)
```

19.6.19 Getting a Definition

The **GET D**efinition command expands the current expression in line if it is one of the following cases:

1. If the CAR of the current expression is the name of a macro, the macro name is replaced by its current expansion.
2. If the CAR of the current expression is a CLISP word, the current expression is replaced by its translation.

```
*P
((if (NLISTP X) then (CAR X)))
*1
*GETD
*PP
(COND
     ((LISTP X)
          (CAR X)))
```

   Note that the translation eliminated the double negative.
3. If CAR of the current expression is a function name, the arguments are substituted into the body of the function definition and the result replaces the function invocation.

```
←(EDITF ROUNDTO)
edit
*PP
(LAMBDA (X Y)
```

```
      (TIMES (ROUNDED (QUOTIENT X Y) Y)))
*3 2 P
(ROUNDED (QUOTIENT X Y) Y)
*GETD
*PP
(TRUNCATE (PLUS (QUOTIENT X Y)
                (QUOTIENT (SIGN (QUOTIENT X Y))
                          2)))
```

GETD must be able to access the symbolic definition of the function from memory or from a noticed file.

4. If CAR of the current expression is LAMBDA, the arguments are substituted into the body of the LAMBDA which replaces the LAMBDA expression in the current expression.

A macro definition for GETD is (from EDITMACROS)

```
(GETD NIL
      UP
      (ORR
          ((I 1 (OR
                    (EDIT GETD (##1)
                          (AND (CDR L)
                               (EDITLO L '!0)))
                    (ERROR!)))
          ((E 'GETD?)))
      1)
```

19.6.20 Getting a Value

The **GET VALue** command replaces the current expression by its value after it has been evaluated. Note that evaluation may generate an error if the evaluation depends on the establishment of a certain environment. It takes the form

```
*GETVAL
```

The current expression in the edit chain is evaluated and the result replaces the expression that was evaluated. You cannot replace the top-level expression in the edit chain with its value.

A macro definition for GETVAL is (from EDITMACROS)

```
(GETVAL NIL
        UP
        (ORR
              ((I 1 (EVAL (##1) '*)))
```

```
            ((E 'GETVAL?)))
        1)
```

19.6.21 Getting a Comment

GET* retrieves the full text of a comment if the current expression is a comment pointer (see Section 16.10). The comment pointer is replaced by the full text.

A macro definition for GET* is (from EDITMACROS)

```
(GET* NIL
      (BIND
          (IF (NEQ (SETQ #1 (GETCOMMENT (##)))
                   (##))
              ((I : #1)
               1)
          NIL)))
```

19.6.22 Going to a PROG Label

The **GO** command makes the current expression be an S-expression immediately after the PROG label. It takes the form

```
*(GO ⟨label⟩)
```

If there is no label in the edit chain having the given name, the Editor prints a warning and waits for a new command.

A macro definition for GO is (from EDITMACROS)

```
(GO (LAB)
    (ORR
          ((← ((*ANY* PROG ASSEMBLE DPROG)
                 --LAB ))
           F LAB (ORR 2 1)
           P)
           ((E
               (PROGN
                    (SETQQ COM LAB)
                    (ERROR!))))))
```

19.6.23 Conditional Listing

The **IF** command allows you to perform conditional editing. The general form is

```
*(IF ⟨expression⟩)
```

In this form, the Editor evaluates ⟨expression⟩. If the result is an error or NIL, the result of the command is an error.

At the top level of the Editor, this form is not particularly useful. However, several editor commands use errors to force sequencing through a number of alternatives. The error is interpreted as an indication to try the next expression. For example, consider the location specification

```
(ATTACH (IF (LISTP (## 2))))
```

which says to find the expression with ATTACH as its CAR if the result of executing the expressions in the edit chain yields a list.

IF may also be used to select between two alternative expressions. It takes the form

```
*(IF ⟨expression⟩ ⟨commands1⟩ ⟨commands2⟩)
```

When ⟨expression⟩ is evaluated, a non-NIL value selects ⟨commands1⟩ for execution, while a NIL value selects ⟨commands2⟩. The IRM notes that this form is equivalent to

```
(COMS (CONS 'COMSQ
            (COND
               ((CAR (NLSETQ (EVAL X)))
                     ⟨commands1⟩)
               (T    ⟨commands2⟩)))))
```

The IRM notes that the expression

```
(IF (READP T) NIL (P))
```

will print the current expression if the read buffer is empty.

In this form ⟨commands[2]⟩ is optional. But, if evaluating ⟨expression⟩ generates an error, then the error will occur.

19.6.24 Inserting into an Expression

INSERT inserts one or more S-expressions before, after, or in place of another S-expression. It takes the form

```
*(INSERT ⟨expr1⟩ ... ⟨exprN⟩ ⟨location⟩ ⟨expression⟩)
```

where ⟨location⟩ is one of the keywords BEFORE, AFTER, or FOR.

Suppose that we have a function ENTER that adds a new member to a list if it is not already there. At the top level, MEMBER searches the elements of the

list to determine if an entry already exists in the list. We would like to add an expression that also checks all sublists of the list. We can do this as follows:

```
←(EDITF ENTER)
edit
*PP
(LAMBDA (ENTRY LST)
     (COND
          ((MEMBER ENTRY LST) LST)
          ((NULL LST) (LIST ENTRY))
          (T (APPEND LST (LIST ENTRY)))))
*P
(LAMBDA (ENTRY LST) (COND & & &))
*3 P
(COND (& ENTRY) (& &) (T &))
*(INSERT ((EQMEMB ENTRY LST) LST) AFTER 2)
*P
(COND (& ENTRY) (& LST) (& &) (T &))
```

where the third element in the above list is the element that we just inserted.

An alternative command, **Insert**, evaluates its arguments and inserts the results into the current expression. It takes the form

```
*(I ⟨command⟩ ⟨expression1⟩ ... ⟨expressionN⟩)
```

I executes the Editor command given by ⟨command⟩. It usually locates the position where the expressions will be inserted. ⟨command⟩ may be an atom or a list, whence it is also evaluated. The ⟨expression[i]⟩ are evaluated and the results inserted into the current expression at the location specific. We could have performed the editing indicated above by the following command:

```
*(I 3 (QUOTE ((EQMEMB ENTRY LST) LST)))
```

If ⟨command⟩ is a list, it will be evaluated. This allows you to specify conditional locations for insertion, as suggested by the following expression:

```
*(I (COND ((NULL FLG) '-1) (T 1)) "APPLES")
```

which places the string "APPLES" in a list associated with the index -1 or 1.

Note that I actually modifies the current expression rather than copying it.

19.6.25 Joining Conditional Expressions

The **JOIN Conds** command merges two neighboring conditional expressions. When creating and editing functions, you sometimes add additional checks after

the function was written. This may result in the proliferation of superfluous COND clauses in your function definition. JOINC allows you to combine two COND expressions into one. However, you must be careful to ensure that the COND expressions are independent, i.e., they test for different conditions both of which may not be true.

```
*PP
(PROG NIL
      (COND
          ((EQUAL Y 10) (PRINT Y)))
      (COND
          ((LESSP Y 20) (PRINT (IPLUS Y 12)))
          (T (IMINUS Y 5))))
*JOINC
*PP
(COND
     ((EQUAL Y 10) (PRINT Y))
     ((LESSP Y 20) (PRINT (IPLUS Y 12)))
     (T (IMINUS Y 5)))
```

Note that JOINC performs (F COND T) first to locate the first COND of the pair to be joined. Thus, you do not have to be properly positioned in order to execute JOINC.

A macro definition for JOINC is (from EDITMACROS)

```
(JOINC NIL
       (F COND T)
       UP
       (BI 1 2)
       1
       (BO 2)
       (2)
       (RO 1)
       (BO 1))
```

19.6.26 Locating an S-expression

The **LoCate** command allows you to explicitly invoke the location operation. It takes the general form

```
*(LC . @)
```

where @ is a location specification.

An alternative form, **LoCate Local**, searches only the current S-expression. Consider the following example:

```
*PP
(PROG NIL
      (COND
          ((NULL (SETQ L (CDR L)))
               (COND
                    (FLAG
                         (RETURN L))))
          ((NULL (CDR (MEMBER (CAR L) (CADR L)))))))
*3 P
(COND & &)
*(COND (LCL RETURN) ) P
(COND
     (FLAG
          (RETURN L)))
```

Two alternative forms, **2ND** and **3RD**, perform two or three LC commands in sequence. They may be defined as edit macros as follows:

```
(2ND X
     (ORR (LC . X)
          (LC . X)))

(3RD X
     (ORR (LC . X)
          (LC . X)
          (LC . X)))
```

19.6.27 Inserting and Removing Left Parentheses

The **Left In** and **Left Out** commands insert and remove left parentheses from the current S-expression, respectively. They take the general form

```
*(LI ⟨n⟩)
*(LO ⟨n⟩)
```

where ⟨n⟩ is the numerical index of the element before which a left parenthesis is to be inserted or removed.

Consider the following list:

```
((Los Angeles 1984) (Moscow 1980) Montreal 1976)
```

The last two atoms should actually be components of a list. We can make them be a list by executing

```
*(LI 3)
*P
((Los Angeles 1984) (Moscow 1980) (Montreal 1976))
```

Note that LI inserts a left parenthesis before the indicated element and a matching right parenthesis at the end of the current S-expression. LI ensures balanced parenthesis pairs.

LO removes the left parenthesis from before the indicated element. It deletes all elements following the ⟨n⟩th element in the list.

```
*(LO 2)
*P
((Los Angeles 1984) Moscow 1980)
```

19.6.28 Lower-Case Conversion

LOWER converts all characters of the current expression to lower case. It takes the form

```
*LOWER
```

An alternative form replaces the current expression with the lower-case expression. It takes the form

```
*(LOWER ⟨expression⟩)
```

It might be defined by an edit macro:

```
(LOWER NIL
       (IF (NLISTP (##))
           UP NIL)
       (I 1 (L-CASE (##)))
```

Consider the following example:

```
*PP
(DEFINEQ
     (FACTORIAL (X)
          (* THIS FUNCTION COMPUTES THE FACTORIAL OF X)
          (COND
               ((ZEROP X) 1)
               (T
                    (ITIMES X (FACTORIAL (SUB1 X)))))
     ))
*4
*P
(* THIS FUNCTION COMPUTES THE FACTORIAL OF X)
*LOWER
(* this function computes the factorial of x)
*OK
```

19.6.29 Iterative Execution

LP, the **LooP** command, repeatedly executes its argument list until an error occurs. It takes the form

```
*(LP . ⟨command⟩)
```

Consider the following example:

```
←(DEFINEQ
     (PRINT.RESPONSE (x)
          (PRINT "The response is")
          (PRINT x)
     ))
(PRINT.RESPONSE)

←(EDITF 'PRINT.RESPONSE)
*(LP F PRINT (N T))                    which places a T at
                                       the end of every PRINT
                                       expression
2 occurrences
*OK

←(PP PRINT.RESPONSE)
(PRINT.RESPONSE
     (LAMBDA (X)
          (PRINT "The response is" T)
          (PRINT X T)
     ))
```

Note that when an error occurs, the ⟨command⟩ cannot be applied to the current expression. LP terminates and prints a message: ⟨n⟩ OCCURRENCES, where ⟨n⟩ is the number of times ⟨command⟩ was successfully executed. The edit chain remains in the state established by the last successful execution of ⟨command⟩.

An alternative form, (LPQ . ⟨command⟩), operates like LP, but it does not print the occurrences message.

It is possible for ⟨command⟩ to force you into an infinite loop. To avoid this problem, LP terminates when the number of successful iterations is equal to MAXLOOP. MAXLOOP is initially set to 30. Since the edit chain reflects the last successful execution, you may continue execution by issuing the REDO command (see Section 28.4.1). Setting MAXLOOP to NIL effectively implies an infinite number of iterations.

19.6.30 Macro Definition

The **Macro** definition command allows you to create new commands which are extensions to the Editor's repertoire. You may also redefine existing commands

via macros whence the macros will take precedence in execution. The format of this command is

```
*(M ⟨atom⟩ . ⟨command list⟩)
```

The Editor defines ⟨atom⟩ to be an Editor command. Whenever ⟨atom⟩ is issued, the commands associated with ⟨atom⟩ are executed. For example, to define a backup-and-print command, we could execute the expression

```
*(M BP BK UP P)
```

where BK, UP, and P are the commands associated with the new command BP.

If a macro already exists with the name ⟨atom⟩, it will be redefined. Macros may use any Editor commands, including other macros, in the definition of new Editor commands.

Many times, you will want to provide arguments to Editor commands to make them as general as possible. An alternative format allows you to define a list form of a macro as follows:

```
*(M (⟨atom⟩) (⟨parameter1⟩ ... ⟨parameterN⟩) .
⟨commands⟩)
```

The Editor defines ⟨atom⟩ as an Editor command taking N arguments. Executing an Editor command of the form

```
(⟨atom⟩ ⟨expression1⟩ ... ⟨expressionN⟩)
```

causes ⟨expression[i]⟩ to be bound to ⟨parameter[i]⟩ and ⟨commands⟩ to be executed. A general form of BP might be defined as

```
*(M (BP) (N) (BK N) UP P)
```

This format spreads the arguments given with the command across the parameters specified by the macro definition. An alternative format allows you to define a nospread macro definition. It takes the form

```
*(M (⟨atom⟩) ⟨parameter⟩ . ⟨commands⟩)
```

When a macro command of the form

```
(⟨atom⟩ ⟨expression1⟩ ... ⟨expressionN⟩)
```

is executed, the arguments ⟨expression1⟩ ... ⟨expressionN⟩ are bound to ⟨parameter⟩ throughout the definition. The command 2ND may be defined as a macro:

```
*(M (2ND) X
          (ORR ((LC . X) (LC . X))))
```

You should note that the atomic commands are independent of the list form commands for both macros and built-in commands. For example, LOWER and (LOWER x) are two independent commands that do not conflict with each other. However, it is not a good idea to define new commands having both atomic and list forms with extremely varying effects.

A macro definition of M is (from EDITMACROS)

```
(M (X . Y)
   (E
         (MARKASCHANGED
              (COND
                   ((LISTP 'X)
                        (CAR 'X))
                   (T 'X))
              'USERMACROS)
         T)
    (ORIGINAL (M X . Y)))
```

19.6.31 Assigning Values to Arguments

MAKE assigns a value to a parameter in the current expression. It takes the form

```
*(MAKE ⟨parameter⟩ ⟨value⟩)
```

Consider the following example (assuming the definition for FACTORIAL that was given in Section 19.6.28):

```
←(FACTORIAL Q6)                    where "1" was intended
                                   instead of "q"
UNBOUND ATOM
Q6
←FIX -1
edit
*?=
X = Q6
*(MAKE X 16)
*OK
32068960256
```

MAKE may be defined by an editmacro as follows:

```
(MAKE (VAR . VALS)
      (COMS (MAKECOM VAR VALS)))
```

19.6.32 Making a Function

MAKEFN converts the current expression to a function. It takes the form

```
*(MAKEFN ((function) . (parms)) (arglst) (N1) (N2))
```

where ⟨function⟩ is the function name and ⟨arglst⟩ is a list of its arguments. The argument names are substituted for the corresponding argument values in ⟨parms⟩. The result of this command becomes the body of the function definition for the function. The current expression is replaced by a call to the function of the form

```
((function) . (parms))
```

Consider the following example:

```
*P
(COND
     ((CAR x)
          (PRINT Y T))
     (T
          (HELP)))
*(MAKEFN (AFN (CAR X) Y) (A B)))
*P
(AFN (CAR X) Y)
*OK

←(PP AFN)
(AFN
     (LAMBDA (A B)
          (COND
               (A (PRINT B T))
               (T (HELP)))))
```

If ⟨arglst⟩ is omitted, MAKEFN creates an argument list using the elements of ⟨parms⟩ if they are literal atoms. Otherwise, it selects argument names from the list (X Y Z A B C ...) while avoiding duplicate argument names.

If ⟨N1⟩ and ⟨N2⟩ are supplied as integer numbers, MAKEFN uses (⟨N1⟩ THRU ⟨N2⟩) to form the body of the function. If ⟨N2⟩ is omitted, then (⟨N1⟩ THRU -1) is used.

MAKEFN may be defined by an editmacro as follows:

```
(MAKEFN (FORM ARGS N M)
     (IF (QUOTE M)
```

```
        ((BI N M)
         (LC . N)
         (BELOW))
        ((IF (QUOTE N)
             ((BI N-1)
              (LC . N)
              (BELOW\))
              ((LI 1)))))
    (E (MAKEFN (QUOTE FORM)
               (QUOTE ARGS)
               (##)
               T)
    UP
    (1 FORM)
    1)
```

19.6.33 Marking and Restoring the Edit Chain

MARK adds the current edit chain to the front of MARKLST. This allows you to save the edit chain for later reference. Consider the following example:

```
←(DEFINEQ
    (NEW.VARIABLE NIL
        (PROG (VARIABLE)
              (SETQ VARIABLE (GENSYM))
              (PUTPROP VARIABLE 'VARIABLE T)
              (RETURN VARIABLE))
    ))
←(EDITF 'NEW.VARIABLE)
edit
*3
*3
*3 P
(GENSYM)
*MARK
*0 0 P
(PROG (VARIABLE) (SETQ VARIABLE &) (PUTPROP VARIABLE & T)
(RETURN VARIABLE))
*←
*P
(GENSYM)
```

An alternative form of MARK takes an atom as an argument. It sets the atom's value to be the current edit chain. It takes the form

```
*(MARK ⟨atom⟩)
```

To return to the most recently marked edit chain, you may use the command ← (left arrow) command. ← sets the edit chain to be (CAR MARKLST). Continuing from above:

```
*0 0                    "move up to SETQ and PROG levels"
*4                      "move down to PUTPROP"
*←                      "go back to (GENSYM)"
*P
(GENSYM)
```

If no MARKs have been performed, MARKLST is NIL and ← will generate an error.

An alternative form, ← ← erases the recent mark from MARKLST after it has been established as the new edit chain. It is equivalent to performing

```
(SETQ MARKLST (CDR MARKLST))
```

You may make the current edit chain become the value of an atom using the \ command. It takes the form

```
*(\ ⟨atom⟩)
```

where ⟨atom⟩'s value was previously established by a MARK command. For complex functions, you may use atoms to store the edit chains at particularly crucial points in the code. Returning to one of those points merely requires executing a \ command. If ⟨atom⟩ is NIL, then the current edit chain is set to NIL.

An alternative form, **P**, takes you back along the edit chain to the last PRINT operation (e.g., one of P, ?, or PP). If the edit chain has not changed since the last printing, the edit chain is restored to the one before that. That is, two chains are always saved. Consider the following example:

```
←(EDITF 'NEW.VARIABLE)
editf
*3
*2 P
(VARIABLE)
*0
*4 P
(PUTPROP VARIABLE 'VARIABLE T)
*(R 'VARIABLE 'VAR)
*\P
(VARIABLE)
```

19.6.34 Moving Expressions

The **MOVE** command allows you to move S-expressions from one place to another within an S-expression you are editing. It takes the form

```
*(MOVE @1 TO ⟨command⟩ @2)
```

⟨command⟩ may be BEFORE, AFTER, or a list command. MOVE operates as follows:

1. Performs (LC @1) to determine the expression to be moved.
2. Performs (LC @2) to determine the destination to which the expression is to be moved.
3. Performs (⟨command⟩ @1), e.g., puts the expressions @1 in the S-expression relative to @2.
4. Deletes the expression @1.

Steps 1, 2, and 3 are performed using the original edit chain. The edit chain is unchanged after the execution of a MOVE. UNFIND is set to the current edit chain after (⟨command⟩ @1) is performed.

Note that @2 cannot specify a location inside @1. An attempt to do so causes an error.

Consider the following example:

```
*P
(hydrogen calcium helium lithium)
*(MOVE 2 TO AFTER 4)
*P
(hydrogen helium lithium calcium)
```

19.6.35 Adding to the End of an Expression

N adds one or more expressions to the end of the current expression. It takes the form

```
*(N ⟨expression1⟩ ... ⟨expressionN⟩)
```

N modifies the structure of the current expression using RPLACD.

If the current expression is not a list, N generates an error.

Consider the following example:

```
*P
(COND
     ((ATOM x) x)
     ((LISTP x) (CAR x)))
*(N (T NIL))
*P
(COND
     ((ATOM x) x)
     ((LISTP x) (CAR x))
     (T NIL))
```

19.6.36 Negating the Current Expression

NEGATE negates the current expression. Its format is

```
*NEGATE
```

It might be implemented by embedding a NOT at the beginning of the current expression via (MDB NOT). However, many expressions, when negated, can be further simplified. NEGATE attempts to simplify the expressions that it operates upon.

```
*P
(OR
     (NULL x)
     (LISTP x))
*NEGATE
*P
(AND X (NLISTP x))
```

NEGATE may be defined by an editmacro as follows:

```
(NEGATE NIL
        UP
        (I 1 (NEGATE (##1)))
        1)
```

19.6.37 Advancing to the Next Expression

NEX advances to the next expression of the type specified by its argument. Its format is

```
*NEX
```

or

```
*(NEX ⟨atom⟩)
```

A macro definition of NEX is (from EDITMACROS)

```
(NEX NIL
     (BELOW ←)
     NX)
```

and, alternatively,

```
(NEX (X)
     (BELOW X)
     NX)
```

The atomic form of NEX is useful when you mark an expression in the edit chain; you may then use NEX to step through the sublists of the current expression.

Another form is **NeXT** which advances to the next expression. NX acts like UP followed by 2. If the current expression is the last expression in the edit chain, NX will generate an error.

Another alternative form, **!NX**, makes the current expression be the next expression at a higher level in the edit chain unless you are already at the top of the edit chain. That is, !NX proceeds through any number of right parentheses to match (close) the current expression.

We can also define a macro, NXP, which moves to the next expression and prints it. It has the definition

```
(NXP NIL
     (ORR (NX)
          (!NX (E (PRIN1 ")" T)
               T))
          ((E
               (PROGN
                    (SETQQ COM -1)
                    (ERROR!)))))
     P)
```

19.6.38 The NIL Command

NIL is usually a no-op, e.g., a no operation command. However, when NIL is preceded by F or BF, it indicates that NIL should be sought in the edit chain.

19.6.39 Finding the Nth Element

NTH finds the nth element of the current expression. It takes the form

```
*(NTH ⟨n⟩)
```

where ⟨n⟩ is not equal to 0. If ⟨n⟩ is greater than zero, it is equivalent to N followed by UP. If ⟨n⟩ is less than zero, it is the nth element from the end of the current expression. If the current expression does not have ⟨n⟩ elements, an error is generated.

A generalized NTH command takes the form

```
*(NTH ⟨command⟩)
```

This form performs (LCL . ⟨command⟩) followed by (BELOW \) followed by UP. That is, it looks for ⟨command⟩ in the edit chain. If the search is unsuccessful, an error is generated and the edit chain remains unchanged.

```
*P
(PROG (& &)
loop
      (COND & &)
      (EDITCOM &)
      (SETQ UNFIND UF)
      (RETURN L))
*(NTH UF)
*P
... (SETQ UNFIND UF) (RETURN L))
```

19.6.40 Exiting the Editor

There are three basic commands for exiting from the Editor. **OK**ay exits from the Editor and saves the edited expression in the atom with which the Editor was invoked.

```
←(EDITV COMPLEXFNS)
edit
*P
(CMULT CDIFFERENCE CMINUS CQUOTIENT ...)
.
.
*OK
COMPLEXFNS
```

STOP exits from the Editor with an error. It is mainly used when you want to exit the Editor without saving the edited expression.

```
←(EDITV COMPLEXFNS)
edit
*P
(CMULT CDIFFERENCE CMINUS CQUOTIENT ...)
.
.
*STOP
```

Saving and Exiting the Editor

SAVE exits from the Editor. The state of the edit chain is saved on the property list of the atom that was edited under the property EDIT-SAVE. If the Editor is called again on the atom, it checks the property list to see if an edit state exists under the property EDIT-SAVE. If so, editing continues from that edit state.

19.6.41 Using the Original Definition

When you issue an Editor command, the Editor determines if a macro has been substituted for the original command. That is, the value of USERMACROS is

searched first for any macro definitions. **ORIGINAL** instructs the Editor to execute all commands which are given as its arguments without considering macro definitions. It takes the form

```
*(ORIGINAL ⟨command1⟩ ... ⟨commandN⟩)
```

19.6.42 Executing Any One Command

ORR begins executing the commands which are its arguments. The first command which completes without an error causes ORR to terminate. It takes the form

```
*(ORR ⟨command1⟩ ... ⟨commandN⟩)
```

Each command is executed in turn. If a command causes an error, ORR restores the edit chain to its original value and begins executing the next command. If all commands execute with errors (e.g., no command completes successfully), then ORR drops off the end of the command list and generates an error.

NIL is a legal command that always executes successfully. Thus, placing NIL at the end of the command list ensures that ORR never generates an error if none of the preceding commands does not generate an error.

```
(ORR NX !NX NIL)
```

attempts to perform an NX, then an !NX, and finally, executes NIL.

19.6.43 Printing the Current Expression

The Editor provides a number of commands that print the current expression. The basic printing command is **P**rint. It takes the form

```
*P
```

which means to print the current expression.

Another form is

```
*(P ⟨m⟩ ⟨n⟩)
```

which means print the Mth element of the current expression as though the print level were set to ⟨n⟩. If ⟨n⟩ is absent, the Editor assumes a print level of 2, e.g.,

```
*(P ⟨m⟩)
```

If ⟨m⟩ is 0, it has the same effect as P. If ⟨m⟩ is 0 and ⟨n⟩ is greater than 0, it has the same effect as P with a print level of ⟨n⟩.

The command **?** is equivalent to (P 0 100).

Several Editor commands perform prettyprinting. The basic command is **PrettyPrint**, which prettyprints the current expression (see Section 15.7).

An alternative form, **PP***, prettyprints the current expression with comments. Note that PP merely prints **COMMENT** which is the value of **COMMENT**FLG (see Section 16.10.1).

Another form, **PPV**, prints the current expression as a variable. That is, it provides no special treatment for LAMBDA, SETQ, PROG, or CLISP expressions.

Finally, **PPT** prettyprints the current expression with any CLISP translations (see Chapter 23). PPT may be defined as an edit macro as follows:

```
(PPT NIL
     (RESETVAR CLISPRETRANFLG T PP))
```

19.6.44 Replacing in an Expression

The **R**eplace command replaces all instances of one expression in the current expression by another. It takes the form

```
*(R ⟨x⟩ ⟨y⟩)
```

where ⟨x⟩ and ⟨y⟩ are S-expressions. If there is no instance of ⟨x⟩ in the current expression, R generates an error.

The expression ⟨x⟩ may be an atom or string. If it contains the character sequence $⟨s⟩, e.g., ⟨ESC⟩⟨s⟩, then $⟨s⟩ appearing in ⟨y⟩ will stand for the corresponding characters matched in ⟨x⟩. You will be informed of all such replacements by a message of the form, ⟨x⟩ -⟩ ⟨y⟩. Note that this feature may also be used to delete characters. (R $1 $) will delete all 1s in the current expression. If ⟨x⟩ does not contain $⟨s⟩, then $⟨s⟩ appearing in ⟨y⟩ refers to the entire expression matched by ⟨x⟩.

This command is often used to replace characters in an expression. An alternative command, **RC**, takes the form

```
*(RC ⟨x⟩ ⟨y⟩)
```

which is equivalent to (R $⟨x⟩$ $⟨y⟩$). This form changes all instances of ⟨x⟩ to ⟨y⟩.

Sometimes, you want to change only one instance of a pattern in an expression. **R1**, having the same form as R, changes exactly one instance. **RC1**, similar to RC, changes just one instance of the character.

REPLACE acts just like R. It takes two forms which are more readable and comprehensible:

```
*(REPLACE @ BY ⟨expression1⟩ ... ⟨expressionN⟩)
*(REPLACE @ WITH ⟨expression1⟩ ... ⟨expressionN⟩)
```

where @ identifies the expression to be replaced.

19.6.45 Raising the Case in an Expression

RAISE raises all the characters in the current expression to upper case. It takes one of two forms:

```
*RAISE
```

or

```
*(RAISE ⟨expression⟩)
```

These may be defined as editmacros as follows:

```
(RAISE NIL
       (IF (NLISTP (##))
           UP NIL)
       (I 1 (U-CASE (##1))))
(RAISE (C)
       (I R (L-CASE (QUOTE C)) (U-CASE (QUOTE C))))
```

19.6.46 Editing Atoms or Strings

REPACK allows you to edit an atom's print name or a string by recursively calling the Editor on the UNPACKed list. If the recursion is exited successfully via OK, the list of atoms is PACKed into an atom or string to replace the one that was edited. The new atom or string is always printed.

REPACK may be defined as an editmacro as follows:

```
(REPACK NIL
        (IF (LISTP (##))
            (1)
            NIL)
        (I :
           ((LAMBDA (X Y)
               (SETQ COM (QUOTE REPACK))
               (SETQ Y
                     (APPLY (QUOTE CONCAT)
                            (EDITE (UNPACK X))))
               (COND
                    ((NOT (STRINGP X))
```

```
                        (SETQ Y (MKATOM Y))))
            (PRINT Y T T))))
    (##))
```

19.6.47 Right Parenthesis In

You may insert a right parenthesis into an expression using the **R**ight parenthesis **I**n command. It takes the form

```
*(RI ⟨n⟩ ⟨m⟩)
```

where ⟨n⟩ and ⟨m⟩ are the indices of elements in the current expression. The right parenthesis of the ⟨n⟩ element is inserted (moved) after the ⟨m⟩ element.

```
←(SETQ presidents
       '((republicans (nixon ford lincoln reagan))
         (democrats (kennedy johnson carter
         (whigs))))
((republicans (nixon ford lincoln reagan) (democrats
(kennedy jonhnson carter) (whigs)))
```

There is a right parenthesis missing after the expression ending in CARTER. We can insert it using the following sequence of commands:

```
*(EDITV presidents)
edit
*(RI 2 2)
*P
((republicans &) (democrats &) (whigs))
```

19.6.48 Right Parenthesis Out

You may remove a right parenthesis from the current expression using the **R**ight parenthesis **O**ut command, which takes the form

```
*(RO ⟨n⟩)
```

where ⟨n⟩ indicates the Nth element of the current expression. The parenthesis is moved to the end of the current expression. All elements following the Nth element are moved inside the Nth element. If the Nth element is not a list, an error will be generated. Using the example given above,

```
←(EDIT presidents)
edit
*P
```

```
((republicans &) (democrats &) (whigs))
*(RO 3)
*(RO 2)
*(RO 1)
*(LO 1)
*(LO 3)
*(LO 5)
*P
(republicans (nixon ford lincoln reagan) democrats
(kennedy johnson carter) whigs)
```

19.6.49 Setting a Literal Atom's Value

You may set a literal atom to the value of the current expression using the **Set** command, which takes the form

```
*S ⟨litatom⟩ @
```

where ⟨litatom⟩ is the name of the atom which will be assigned the current expression after performing (LC . @).

```
←(EDITF CMULT)
edit
*P
(LAMBDA (CX1 CX2) **COMMENT** (PROG & & &))
*(S JUNK -1 4)
*STOP

←JUNK
(RETURN CX3)
```

19.6.50 Showing Instances

SHOW displays all of the instances of a given pattern in the current expression. It takes the form

```
*(SHOW ⟨pattern⟩)
```

Consider the following example:

```
←(EDITF initialize.frame)
edit
*(SHOW PUTVALUE)
(PUTVALUE NAME (QUOTE TYPE) NIL)
(PUTVALUE NAME (QUOTE SUPER) NIL)
```

```
(PUTVALUE NAME (QUOTE DESCRIPTION) NIL)
done
```

SHOW may be defined as an edit macro as follows:

```
(SHOW X
      (F (*ANY* . X) T)
      (LPQ MARK (ORR 1 !0) NIL)
           P ← ←
           (F (*ANY* . X) N))
      (E (QUOTE done)))
```

19.6.51 Splitting Conditional Expressions

SPLIT Conds is a command that splits a conditional expression into two conditional expressions. It takes the form

```
*(SPLITC ⟨x⟩)
```

where ⟨x⟩ specifies the last clause in the first conditional (after the split occurs). Using the example from JOINC above, consider the following:

```
*PP
((COND
     ((EQUAL Y 10) (PRINT Y))
     ((LESSP Y 20) (PRINT (IPLUS Y 12)))
     (T (IMINUS Y 5))))
*(SPLITC 2)
*PP
((COND
     ((EQUAL Y 10) (PRINT Y)))
 (COND
     ((LESSP Y 20) (PRINT (IPLUS Y 12)))
     (T (IMINUS Y 5))))
```

SPLITC may be defined as an editmacro as follows:

```
(SPLITC (X)
     (F COND T)
     (BI 1 X)
     (IF (AND (EQ (##2 1) T)
              (##2 2)
              (NULL (CDDR (##)))
         ((BO 2)
          (2))
```

```
        ((-2 COND) (LI 2)))
    UP
    (BO 1))
```

19.6.52 Switching Elements in an Expression

You may switch elements in the current expression using the **SWitch** command. It takes the form

```
*(SW ⟨n⟩ ⟨m⟩)
```

where ⟨n⟩ and ⟨m⟩ indicate the Nth and Mth elements to be switched, respectively. Consider the following example (after the IRM):

```
*PP
(LIST (CONS (CAR X) (CAR Y))
      (CONS (CDR X) (CDR Y)))
*(SW 2 3)
*P
(LIST (CONS (CDR X) (CDR Y))
      (CONS (CAR X) (CAR Y)))
```

Note that the relative order of ⟨n⟩ and ⟨m⟩ is not important; (SW 3 2) would have worked equally as well.

An alternative form, **SWAP**, switches expressions. It takes the form

```
*(SWAP @1 @2)
```

where @1 and @2 are location specifications. Thus, using the original expression above,

```
*PP
(LIST (CONS (CAR X) (CAR Y))
      (CONS (CDR X) (CDR Y)))
*(SWAP CAR CDR)
*P
(LIST (CONS (CDR X) (CAR Y))
      (CONS (CAR X) (CDR Y)))
```

SWAP may be defined as an edit macro as follows:

```
(SWAP (LC1 LC2)
     (BIND (MARK #3)
           (LC . LC1)
           (MARK #1)
```

```
(\ #3)
(LC . LC2)
(MARK #2)
(IF
    (NOT (OR
              (MEMB (CAR #1) #2)
              (MEMB (CAR #2) #1)))
    (UP (\ #1) UP
     (I 1 (CAR #2))
     (\ #2)
     UP
     (I 1 (CAR #1)))
    (E (QUOTE (nested expressions)))
(\ #3)))
```

Another alternative form, **SWAP** Cond clauses, swaps clauses within a conditional expression. It takes the form

```
*(SWAPC ⟨n⟩ ⟨m⟩)
```

where ⟨n⟩ and ⟨m⟩ indicate the COND clauses to switch. Using the example from SPLITC, we have

```
*PP
((COND
     ((EQUAL Y 10) (PRINT Y))
     ((LESSP Y 20) (PRINT (IPLUS Y 12)))
     (T (IMINUS Y 5))))
*1
*(SWAPC 2 3)
*PP
(COND
     ((LESSP Y 20) (PRINT (IPLUS Y 12)))
     ((EQUAL Y 10) (PRINT Y))
     (T (IMINUS Y 5)))
```

Note that SWAPC assumes that the current expression is a COND expression. Remember that the first element is the COND atom.

19.6.53 Setting a Tentative Edit Marker

TEST adds an undo-block at the beginning of the UNDOLST for the Editor. It provides a marker that allows you to perform a number of editing changes, and then undo them all with a single command, !UNDO.

UNBLOCK removes an undo-block from the UNDOLST.

See the description of uNDO for an explanation of their usage.

19.6.54 THRU and TO: Location Specification

The **THRU** command is used to specify a collection of expressions that can be treated as a single element. It is used by EXTRACT, EMBED, DELETE, REPLACE, and MOVE. It takes the form

```
*(@1 THRU @2)
```

where @1 and @2 are location specifications. THRU performs (LC . @1), UP, (BI 1 @2), 1. This has the effect of grouping the desired expressions into a single expression which may be treated by another command.

The **TO** command has the same form and effect except that the last expression is not included in the grouping. It performs the sequence (LC . @1), UP, (BI 1 @2), (RI 1 -2), 1.

If both @1 and @2 are numbers, and @2 is greater than @1, @ counts from the beginning of the current expression. For example, using the following list (A B C D E F G), (3 THRU 5) means (C THRU E) while (3 TO 5) means (C TO E).

THRU and TO do not do anything useful by themselves. They are very powerful when combined with other commands as noted above.

19.6.55 Recursive Editing

The **TTY:** command allows you to call the Editor recursively from within the Editor. You exit from the lower Editor via OK or STOP. All Editor commands are available in the recursed version of the Editor including TTY:. The edit chain is preserved across recursive calls to the Editor.

19.6.56 Undoing an Editor Command

The Editor maintains an internal list, UNDOLST, which records each change to the structure of the current expression by the structure modification commands. An Editor command's effects may be undone using the **UNDO** command. It takes the form

```
*UNDO
```

This command undoes the effects of the most recent structure modification command, e.g., the last entry on UNDOLST. After the effect has been undone, it prints the name of the command which was undone. The edit chain is restored to the state prior to the undone command. If there are no commands to undo, the Editor types NOTHING SAVED.

An alternative form, **!UNDO**, undoes all structural changes made during the current editing session. As each command is undone, its name is printed. It acts like UNDO.

Note that if a SAVE was performed between calls to the Editor, only changes made since the last invocation of the Editor will be undone, i.e., up to

the last SAVE. These changes are protected by inserting a marker, called an *undo-block*, into the front of UNDOLST before editing begins in the new session. UNDO (respectively !UNDO) will undo commands up to the marker.

Note that TEST and UNBLOCK allow you to dynamically insert into and remove from UNDOLST your own undo-block markers.

Undoing changes associated with the I, E, or S commands also undoes the side effects associated with the evaluation of the arguments of those commands.

19.6.57 Moving Up the Edit Chain

The **UP** command allows you to move up the edit chain to the next higher expression. It takes the form

```
*UP ⟨command⟩
```

where ⟨command⟩, an optional argument, indicates that you want to move up to the next higher instance of that particular command.

```
*PP
(COND
     ((LESSP Y 20)
          (PRINT (IPLUS Y 12)))
     ((EQUAL Y 10)
          (PRINT Y))
     (T
          (IMINUS Y 5)))
*(F PRINT T)
*P
(PRINT (IPLUS Y 12))
*UP
... (PRINT &))
```

19.6.58 Extracting from the Current Expression

You may extract a subexpression from within the current expression using the e**XTR**act command. It takes the form

```
*(XTR . @)
```

where @ is a location specification.

XTR replaces the original current expression with the expression that is extracted from it. The new current expression is identified by performing (LCL . @).

```
*PP
((COND
```

```
     ((NULL X)
          (PRINT Y))))
*(XTR 2 2)
*P
((PRINT Y))
```

An alternative form, **EXTRACT**, takes the form

```
*(EXTRACT @1 FROM . @2)
```

where @1 and @2 are location specifications. Thus, we could have specified the above operation as

```
*(EXTRACT PRINT FROM COND)
*P
((PRINT Y))
```

19.6.59 Inserting Comments

You may insert a comment into the current expression using the * command, which takes the form

```
*(* . ⟨text⟩)
```

where ⟨text⟩ is the text of the comment.

The * command ascends the edit chain looking for a safe place to insert the comment. These places include after a COND clause, after a PROG clause, after the arguments to a LAMBDA expression. It will insert the comment after the location if possible; otherwise, before it.

```
*PP
(COND
     ((LESSP Y 20)
          (PRINT (IPLUS Y 12)))
     ((EQUAL Y 10)
          (PRINT Y))
     (T
          (IMINUS Y 5)))
*3 P
((EQUAL Y 10) (PRINT Y))
*(* test Y)
*PP*
((EQUAL Y 10)                                   (* test Y)
     (PRINT Y))
```

19.6.60 Attention-Changing Commands

Several commands allow you to change the current expression by shifting the focus of the Editor. These commands change the edit chain. These commands take the form

```
*⟨n⟩
*-⟨n⟩
```

where ⟨n⟩ is the index of an element within the current expression.

If ⟨n⟩ is 0, the edit chain is set to the CDR of the edit chain, thus moving to the next higher expression. If there is no higher expression, the Editor generates an error.

An alternative form, **!0**, performs repeated 0s until it reaches a point where the current expression is not a tail of the next higher expression. That is, it always goes back to the next higher left parenthesis.

We can define a few simple editmacros that change attention and print the result:

```
(-1P NIL
     (ORR -1 P)
     ((E
          (PROGN
               (SETQQ COM -1)
               (ERROR!))))
(2P NIL
    (ORR (2)
         (1)
         ((E
               (PROGN
                    (SETQQ COM 2)
                    (ERROR!)))))
```

20

Debugging Facilities

INTERLISP provides a set of integrated facilities for analyzing and modifying programs when errors occur. Together, these facilities are known as the *Break Package*. Components of the Break Package are called whenever an error occurs. They may also be invoked by you through the judicious execution of several functions.

There are three possible modes in which you may initiate a break when your program is operating (more or less) normally:

1. *Trace* mode where you may view the arguments passed to and results returned from a function each time it is computed.
2. *Break* mode where you may specify a function is to be interrupted each time it is invoked.
3. *Breakin* mode where you may insert a breakpoint inside an expression that will cause it to be interrupted whenever that expression is executed.

In the latter two modes, the break commands discussed in Section 20.2 may be executed to inspect the state of the computation, change the function definition or variable values, and restart the computation.

20.1 TRACE: TRACING A FUNCTION

TRACE allows you to identify a function that will be "traced" each time it is invoked. It takes the form

```
Function:       TRACE
# Arguments:    1
Argument:       1) an expression, EXPRESSION
Value:          The name of the function to be traced.
```

TRACE is an NLAMBDA, nospread function. By tracing, we mean the following:

1. When the function is invoked, its parameters are printed with their identifying atom names. This action is performed after the values of the arguments have been determined (subject to the function type — see Section 8.1).
2. The value of the function is computed.
3. The result of executing the function is printed.

TRACE accepts a single argument that is either an atom or a list of expressions describing the functions to be traced and the parameters to be printed.

The simplest format for invoking TRACE appears as

```
(TRACE ⟨function⟩)
```

where ⟨function⟩ may be any system- or user-defined function specified as an atom. TRACE executes

```
(BREAKO ⟨function⟩ T '(TRACE ?= NIL GO))
```

The ⟨function⟩ may not be EVAL. Suppose you tried to trace EVAL. EVAL will be called from the READ phase of the top-level loop to evaluate the TRACE command. It calls TRACE, which modifies EVAL with the tracing information. However, to display a trace requires evaluation, which means that EVAL enters an infinite loop while trying to trace itself.

Suppose we trace the functions REAL, IMAG, ITIMES, PLUS, DIFFERENCE, and COMPLEX which are called from within the function CMULT. We would specify tracing for REAL as follows:

```
←(TRACE REAL)
(REAL)

...

←(SETQ cx1 (COMPLEX 2.0 4.0))
((2.0 . 4.0))
←(SETQ cx2 (COMPLEX 3.0 5.0))
((3.0 . 5.0))
←(CMULT cx1 cx2)
REAL:
CX = ((2.0 . 4.0))
REAL = 2.0
```

```
REAL:
CX = ((3.0 . 5.0))
REAL = 3.0

ITIMES:
*ARG1* = 2.0
*ARG2* = 3.0
U = 2
ITIMES = 6

... (e.g., more computations traced)

COMPLEX:
R = -14
I = 22
COMPLEX = ((-14.0 . 22.0))
```

When REAL is traced, the name of its argument is printed and followed by its value. ITIMES, because it is a hardwired function, does not have explicit variable names, so two are manufactured by the Break Package.

Alternatively, EXPRESSION may be a list of the form

```
(⟨function⟩ ⟨variable-list⟩)
```

The CAR of EXPRESSION is the name of the function to be traced, and the CDR is a list of variables that are to be printed when the function is traced. TRACE executes the expression

```
(BREAK0 (CAR ⟨expression⟩)
T
(LIST 'TRACE '?= (CDR ⟨expression⟩) 'GO))
```

Suppose I wanted to see only the value of R for the function COMPLEX. I would set up the trace as follows:

```
←(TRACE (COMPLEX R))
(COMPLEX)

←(CMULT cx1 cx2)
COMPLEX:
R = -14.0
COMPLEX = ((-14.0 . 22.0))
```

If you want to see only the value of the function after it has been executed, you may use the form

```
(⟨function⟩)
```

which causes TRACE to execute the expression

```
(BREAKO ⟨function⟩ T '(TRACE ?= (NIL) GO))
```

In general, output generated by tracing functions is sent to your terminal. The destination of output is controlled by the system variable BRKFILE whose value is initially T. If you want to redirect tracing information to an external file, you must set BRKFILE to the name of that file. You are responsible for opening the file for output.

A Definition for TRACE

We might define TRACE as follows:

```
(DEFINEQ
    (trace
        (NLAMBDA x
            (MAPCONC
                (COND
                    ((ATOM x)
                        (*
                            Trace a single
                            function whose name is
                            given.
                        )
                        (LIST x))
                    (T x))
                (FUNCTION TRACE-FN))
    )))
(DEFINEQ
    (trace-fn (fn)
        (PROG (brkcoms)
            (COND
                ((OR (ATOM fn)
                    (EQ (CADR fn) 'IN))
                    (*
                        Trace a function modified
                        by ADVISE.
                    )
                    (SETQ brkcoms
                        '(TRACE ?= NIL GO)))
                (T
                    (SETQ brkcoms
                        (LIST 'TRACE
                            '?=
```

```
                                    (OR (CDR fn) 'NIL)
                                    'GO))
                            (SETQ fn (CAR fn))))
                (RETURN
                    (BREAKO fn T brkcoms)))
    ))
```

20.2 BREAK COMMANDS

You interact with the Break Package to determine the status of your function's environment via a set of Break Commands. You may also use the stack access and manipulation functions described in Chapter 30.

20.2.1 Releasing Breaks

Perhaps the very first thing you need to know is how to get out of a break once your function has been broken into. Releasing a break allows a computation to proceed. You may use either

GO	Evaluates BRKEXP and prints its value.
OK	Evaluates BRKEXP but does not print its value.

BREAK1 (see Section 20.3) evaluates BRKEXP which was set up by the function that invoked BREAK1. The value of BRKEXP is returned as the value of the function that was broken.

```
←(BREAK MAKESLOT)                "declares MAKESLOT breakable"
(MAKESLOT)

←(MAKESLOT 'TEXAS 'CAPITOL 'IS 'SLOT)
(MAKESLOT broken)
:GO
MAKESLOT = CAPITOL
CAPITOL
```

where MAKESLOT returns the name of the slot as its value. In this case, BRKEXP was merely the name of the function to be broken. Thus, executing BRKEXP means evaluating the function and printing its value.

The same example using OK would appear as

```
←(BREAK MAKESLOT)                "declares MAKESLOT breakable"
(MAKESLOT)
```

```
←(MAKESLOT 'TEXAS 'CAPITOL 'IS 'SLOT)
(MAKESLOT broken)
:GO
CAPITOL
```

Alternative Versions of GO and OK
Alternative forms are !GO and !OK. They act as follows:

1. Unbreak the function
2. Evaluate the function
3. Rebreak the function
4. Execute either GO or OK

That is, they are equivalent to !EVAL followed by either GO or OK.

20.2.2 Evaluation in a Break

You may evaluate the broken function but maintain the break by issuing the EVAL command. This command is useful when you have identified a function to be broken but are not sure of the value it will produce.

```
←(MAKESLOT 'TEXAS 'CAPITOL 'IS 'SLOT)
(MAKESLOT broken)
:EVAL
MAKESLOT evaluated
:!VALUE
CAPITOL
```

The Break Package stores the value of the function in the variable !VALUE. You may see the value by specifying !VALUE as shown above. Using EVAL, you can see what value the function produces. If it is not the correct value, you may invoke the editor to modify the function and, then, re-evaluate it.

Alternative Form of EVAL
An alternative form of EVAL is !EVAL, which

1. Unbreaks the function
2. Evaluates BRKEXP
3. Rebreaks the function

This form is useful for dealing with recursive functions. Evaluating a recursive function inside a break will cause the function to be broken again. Breaks are nested so that the most recent break is attended to first. Thus, !EVAL maintains the original break while allowing you to evaluate the function.

20.2.3 Returning a Value from a Break

After a function is broken, you may force the value of the function to be different from what would actually be computed by it. The RETURN command has two forms:

`RETURN ⟨expression⟩`	`Evaluates the expression and returns its value.`
`RETURN (⟨fn⟩ ⟨arguments⟩)`	`Executes the function with the given arguments and returns its value.`

The RETURN command is useful in a number of ways:

1. When the function broken is returning the wrong value, you force the right value in order to continue testing your program and repair the function later.
2. When the function returns a correct value, you break it and force an erroneous value in order to test the error handling features of your program.
3. When you are programming using a top-down methodology and the function broken is merely a stub. In order to test the logic of higher functions, you force the function to return a value like that it would have computed in order to allow the testing to proceed.

Consider the folowing example:

```
←(MAKESLOT 'TEXAS NIL NIL NIL)
(MAKESLOT broken)
:RETURN 'CAPITOL
MAKESLOT = CAPITOL
CAPITOL
```

where I failed to provide the name of the slot, but I forced the evaluation to act as if I had.

RETURN is also useful for experimenting with the possible values a function might return. You might apply a function to !VALUE after EVALuating a broken function.

20.2.4 Aborting a Break

You may abort a break by issuing the ↑ (up arrow) command which calls ERROR! (see Section 18.6.2). ↑ effectively erases the stack frame for the broken function. This command is useful for unwinding from one or more nested breaks.

Consider the function CMULT which invokes the functions REAL and IMAG. Suppose I specify breakpoints on CMULT and REAL as follows:

```
←(BREAK CMULT)
(CMULT)

←(BREAK REAL)
(REAL)

←(CMULT cx1 cx2)
(CMULT broken)
:GO
(REAL broken)              "here is the second break"
:↑
(CMULT broken)             "here we have returned to the
                            first breakpoint"
...
```

20.2.5 Unbreaking a Function

You may unbreak a function by issuing the UB command:

```
←(BREAK MAKESLOT)
(MAKESLOT)

←(MAKESLOT 'TEXAS 'CAPITOL 'IS 'SLOT)
(MAKESLOT broken)
:UB
(MAKESLOT)
:GO
MAKESLOT = CAPITOL
```

UB removes the effect of the BREAK function (see Section 20.3.1) which modifies the function to break whenever it is called.

20.2.6 Displaying Arguments and Bindings

When a break occurs, you are placed in the Break Package at the current stack frame. One of the first things you might want to do is to inspect the bindings of the arguments since this is a frequent source of trouble when errors occur. If you don't remember the arguments, you can obtain their names via another command. You may also see where a variable is bound in the stack and by which functions. These commands are

```
?=                         Prints the argument names and their
                           current values. ?= looks for
```

	additional arguments on the input line. If the line is empty, ?= prints all of the arguments of the broken function. The remainder of the input line may contain the names of specific arguments to be printed or expressions operating upon the values of those arguments. The values are printed after the expressions have been evaluated.
ARGS	Prints the names of variables bound at the current stack frame.
PB ⟨variable⟩	Prints the bindings of the specified variable for each frame in which it is found and the name of the frame. PB ascends the stack from stack frame LASTPOS.

Consider the following examples:

```
←(SETQ ASLOT 'CAPITOL)
CAPITOL

←(BREAK MAKESLOT)
(MAKESLOT)

←(MAKESLOT 'TEXAS ASLOT 'IS 'S)
(MAKESLOT broken)
:?=
NODE = TEXAS
SLOT = CAPITOL
INHERITANCE.TYPE = IS
METHOD = S

:ARGS
(NODE SLOT INHERITANCE.TYPE METHOD)

:PB NODE
@MAKESLOT : TEXAS
@TOP : NOBIND
```

The latter line says that NODE is not bound outside of the function call. We may also ask where ASLOT is bound (it was provided as an argument to MAKESLOT).

```
:PB ASLOT
@TOP : CAPITOL
```

which says ASLOT is a global variable not bound by any function.

Finally, let us ask for the bindings for LISPXID which is used by LISPX (see Section 25.2):

```
:PB LISPXID
@LISPX : ←
@EVALQT : ←
@TOP : NOBIND
```

which shows that LISPXID is bound by two functions, LISPX and EVALQT, to the same value. Even so, there are two bindings on the stack.

?= may take arguments that specify the variables it is to work on.

```
←(BREAK MAKEDEMON)
(MAKEDEMON)
←(MAKEDEMON 'TEXAS
            'COUNT-CITIES
            '(LAMBDA NIL
                  (COND
                       ((NULL (GETVALUE NODE 'CITY))
                            0)
                       (T
                           (LENGTH
                             (GETVALUE NODE 'CITY)))))
            'DO
            'S)
(MAKEDEMON broken)
:ARGS
(NODE DEMON FUNCTION INHERITANCE.TYPE METHOD)
:?= (CAR FUNCTION)
(CAR FUNCTION) = LAMBDA
```

where we extract the CAR of the value of the specified argument. This is equivalent to saying

```
:(CAR FUNCTION)
LAMBDA
```

The difference is that ?= always evaluates its arguments with respect to the current stack frame. Thus, using ?= we can evaluate a name at any point in the stack if we have previously changed our focus to a different stack frame.

?= also accepts numbers as indices to the argument list. Floating point numbers are truncated to the nearest integer.

```
:?=2
DEMON = COUNT-CITIES

:?=2.4
DEMON = COUNT-CITIES
```

Note that ?= prints each variable and its value on a separate line. This printing is controlled by BREAKDELIMITER, which initially has the value ⟨CR⟩. To modify the appearance of its output, set BREAKDELIMITER to an appropriate string.

Values printed by ?= and PB are printed by SHOWPRINT (see Section 15.1.3). To prettyprint these values, set SYSPRETTYFLG to T.

20.2.7 Obtaining a Backtrace

When you enter a broken function, one of the first things you may want to do is to inspect the history of how the function was called. To do so, you obtain a backtrace by issuing the BT command. BT prints a backtrace of function names only starting at the current position. Typically, this will be the stack frame of the broken function (although you may reset it by defining a different value for LASTPOS).

```
←(MAKESLOT 'TEXAS 'CAPITOL 'IS 'SLOT)
(MAKESLOT broken)
:BT
MAKESLOT
\EVALFORM
\SAFEEVAL
BREAK1
\EVALFORM
EVAL
LISPX
ERRORSET
EVALQT
\REPEATEDLYEVALQT
\EVALFORM
ERRORSET
\MAKE.PROCESSO
T
```

This example is taken from a Xerox 1100 Scientific Information Processor executing the Fugue release of INTERLISP-D.

BTV displays the stack frame names and the associated with their functions:

```
←(MAKESLOT 'TEXAS 'CAPITOL 'IS 'SLOT)
(MAKESLOT broken)
:BTV
MAKESLOT
     *FORM*          (MAKESLOT)
     *ARGVAL*        NIL
     *TAIL*          NIL
     *FN*            MAKESLOT
\EVALFORM
     \INTERNAL       BREAK-EXP
\SAFEEVAL
     BRKEXP          (MAKESLOT)
     BRKWHEN         NIL
     BRKFN           NIL
     BRKCOMS         NIL
     BRKTYPE         NIL
     ERRORN          NIL
BREAK1
     *FORM*          (BREAK1 (MAKESLOT))
\EVALFORM
     \INTERNAL       ←
EVAL
     LISPXID         ←
     HELPCLOCK       507697
     LISPXHIST       ((&) ← ')
     HELPFLAG        T
     LISPXLISTFLG    T
     LISPXLINE       NIL
     LISPY           NIL
     LISPZ           NIL
     LISPXVALUE      NIL
     LISPXTEM        93
LISPX
ERRORSET
     LISPXID         ←
EVALQT
\REPEATEDLYEVALQT
     *FORM*          (\PROC.REPEATEDLYEVALQT)
     *ARGVAL*        NIL
     *TAIL*          NIL
     *FN*            \PROC.REAPEATEDLYEVALQT
...
```

where ... indicates that there is additional information necessary to set up the virtual machine environment on the Xerox 1100 which does not concern us. Variables of the form *⟨name⟩* are called *blipvalues* (discussed in Section 30.2.3).

Usually, this information should be enough for any INTERLISP applications programmer to examine and modify his or her functions. Indeed, using BTV and the stack functions, you may explore the various underlying aspects of a particular implementation. You should exercise caution, however, since many of the internal functions will change between releases as improvements are made to the system.

Using BTV+, you obtain not only the function variables, but also the local variables used by the functions:

```
:BTV+
      *local*          MAKESLOT
      *local*          1
      *local*          MAKESLOT
      *local*          4
      *local*          0
      *local*          (LAMBDA
                            (NODE SLOT INHERITANCE.TYPE
                            METHOD)
                            (BREAK1 & T MAKESLOT NIL))
      *local*          (NODE SLOT INHERITANCE.TYPE METHOD)
      *local*          4
      *local*          8
      *local*          17
      *local*          METHOD
      *local*          NIL
      *local*          NIL
      *local*          NIL
MAKESLOT
      *FORM*           (MAKESLOT)
      *local*          (LAMBDA
                            (NODE SLOT INHERITANCE.TYPE
                            METHOD)
                            (BREAK1 & T MAKESLOT NIL))
      *ARGVAL*         NIL
      *TAIL*           NIL
      *FN*             MAKESLOT
\EVALFORM
      *local*          (MAKESLOT)
```

... you get the idea. You may see how some of the local variables are used by inspecting the code for MAKESLOT when it is compiled.

BTV* operates exactly like BTV except that it prints arguments to SUBRs, local variables, and temporaries of the interpreter.

BTV* produces the exact same display as BTV+ under INTERLISP-D. Note that INTERLISP-D has no SUBRs.

You should note that the numbers displayed by BTV+ and BTV* are printed as octal numbers.

BTV, BTV+, and BTV* all take an optional expression that is a functional argument that allows you to skip stack frames when displaying the contents of the stack.

20.2.8 Displaying the Entire Stack

You may display the entire contents of the stack by issuing the BTV! command. This command displays all basic frames and their frame extensions (see Section 30.1) on the stack.

```
:BTV!

Basic frame at 40174
40160:          0    51655     NODE       TEXAS
40162:          0    52021     SLOT       CAPITOL
40164:          0    15071     INHERITANCE.TYPE    IS
40166:          0      113     METHOD     S
40170:          0    51513     *local*    MAKESLOT
40172:          0        0     [padding]
40174:     100400    40160

Frame xtn at 40176, frame name = MAKESLOT
40176:     141002    40130     [V, USE = 2, alink]
40200:     103524    20040     [fn header]
40202:      40332      412     [next, pc]
40204:      40300    13427     [nametable]
40206:     177777        0     [blink, clink]
40210:         16        5     *local*    5
40212:          0    51513     *local*    MAKESLOT
40214:         16        4     *local*    4
40216:         16        4     *local*    4
40220:          4   135406     *local*    (LAMBDA & &)
40222:         10   135272     *local*    (NODE   SLOT
                               IHERITANCE.TYPE METHOD)
40224:         16        0     *local*    0
40226:         16       10     *local*    8
40230:         16       15     *local*    13
40232:          0    15714     *local*    METHOD
40234:          0        0     *local*    NIL
40236:          0        0     *local*    NIL
```

```
40240:      0        0     *local*   NIL
40242:  177777       0     *local*   [unbound]
        through 40264 repeated
40266:  177777       0     [padding]
40270:      0    51513     [padding]
40272:      0        0     [padding]
40274:  177776       0
40276:  177763      30
40300:      0        0     NIL
40302:      0        0     NIL
40304:      0    51513     MAKESLOT
40306:     10    13400     ([]#0,77400, []#60,174060)
40310:  13621    51604
40312:  51547    15714
40314:      0        0     NIL
40316:      0        0     NIL
40320:      0        0     NOBIND
40322:      2        3     [VMEMPAGEP]#2,3
40234:      0        0     NIL
40236:      0        0     NIL
40330:      0        0     NIL
```

20.2.9 Setting the Stack Frame

The break command @ sets the value of the variable LASTPOS. LASTPOS indicates the stack frame to be used for evaluation, and thus establishes the context for several break commands. LASTPOS is the position of a function call on the stack.

@ resets LASTPOS to (STKNTH -1 'BREAK1). @ may take several arguments that appear on the remainder of the input line. These include

@	Leaves LASTPOS as it was and searches from that point in the stack.
/	Following an atom, it indicates that the next atom is a number which specifies that the preceding atom should be searched for that many times. Thus, CONS is the atom preceding / and if it is followed by 3, CONS will be searched for three times on the stack.
=	Resets LASTPOS to value of the expression which follows in the input line. This allows you to compute the name of a function dynamically and then search for it.

⟨n⟩	⟨n⟩ is a number that indicates how many stack frames up or down the stack you should move in setting LASTPOS.

The action of @ is to process each atom appearing in the remainder of the input line as a direction to search the stack. When the end of the input line is reached, LASTPOS will point to a stack frame to be used for evaluating further commands.

@ will fail with the error message "⟨fn⟩ NOT FOUND" if a stack frame bearing the name ⟨fn⟩ is not found in the stack.

Another command, REVERT, uses LASTPOS to specify a function that is to be re-entered with the arguments found in the stack. Thus, after an error occurs, you may inspect and modify the arguments on the stack, and then re-enter the function with the modified arguments. This provides a mechanism for restarting a computation where an error is discovered below where it actually occurred. You may fix the bug and proceed with the computation.

20.2.10 Setting Values on the Stack

Many errors in INTERLISP programs occur because of unbound atoms or undefined functions. You may dynamically correct these errors by assigning values within a breakpoint.

The = command allows you to assign a value to an unbound atom. It takes the form

```
:= ⟨expression⟩
```

where ⟨expression⟩ sets the unbound atom appearing at that point in the stack frame to the value of ⟨expression⟩. = exits the breakpoint with the assigned value and proceeds with the computation.

The -⟩ command allows you to replace the expression that caused an error with the value of its argument. It takes the form

```
:-⟩ ⟨expression⟩
```

where ⟨expression⟩ is evaluated. This command may be used with either unbound atom or undefined function errors. The value of the expression may be an S-expression, perhaps appearing as the argument of QUOTE.

20.2.11 Breakmacros

A *breakmacro* is a definition for a command that is recognized by BREAK1. When a command is entered, BREAK1 checks to see if it is a breakmacro. The system variable BREAKMACROS is a list of all breakmacro definitions. In the Fugue release, it has the following structure:

```
((DBT!
    (DISPBAKTRACE
         (WINDOWPROP (TTYDISPLAYSTREAM) 'BREAKPOS)
         NIL NIL))
 (DBT
    (DISPBAKTRACE
         (WINDOWPROP (TTYDISPLAYSTREAM) 'BREAKPOS)
         NIL
         '(DUMMYFRAMEP)))
 (BT!
    (BAKTRACE LASTPOS
              NIL
              (BREAKREAD 'LINE)
              0
              T))
 (BTVPP
    (PROG ((SYSPRETTYFLG T))
          (BAKTRACE LASTPOS
                    NIL
                    (CONS 'DUMMYFRAMEP
                          (BREAKREAD 'LINE))
                    1
                    T)))
 (BT
    (BAKTRACE LASTPOS
              NIL
              (BREAKREAD 'LINE)
              0 T))
 (BTV
    (BAKTRACE LASTPOS
              NIL
              (BREAKREAD 'LINE)
              1
              T))
 (BTV*
    (BAKTRACE LASTPOS
              NIL
              (BREAKREAD 'LINE)
              7
              T))
 (BTV+
    (BAKTRACE LASTPOS
              NIL
              (BREAKREAD 'LINE)
              5
              T)
```

```
(BTV!
    (BAKTRACE LASTPOS
              NIL
              (BREAKREAD 'LINE)
              39
              T)))
```

Note: DBT! and DBT are macros relating to the use of the display facilities associated with the Xerox 11xx series computer systems.

20.2.12 Breakresetforms

INTERLISP provides many system variables that affect the way the interpreter and various functions act. You may modify the action of these functions by resetting these variables. When a break occurs, if the variables retain their modified values, debugging may be difficult to perform.

BREAKRESETFORMS contains expressions that are used in conjunction with RESETFORM or RESETSAVE (see Section 25.7). When a break occurs, BREAK1 evaluates each expression on BREAKRESETFORMS before any interaction with the terminal. The values of each expression are saved for later reinstatement. When you exit the Break Package, the state of the system is restored by reinstating the values of the expression appearing on BREAKRESETFORMS. If BREAK1 is re-entered before you exit the Break Package, the expressions on BREAKRESETFORMS are again evaluated and their values saved.

The initial value of BREAKRESETFORMS is:

```
((INTERRUPTABLE T)
 (SETREADMACROFLG)
 (CONTROL)
 (ECHOMODE T))
```

20.3 SETTING BREAKPOINTS

Functions are not actually *broken* until they are invoked. You may set a breakpoint for a function using either BREAK or BREAK0. To set a breakpoint, INTERLISP modifies a function so that it calls BREAK1 upon entry. An example of a function having a breakpoint is

```
(latlon.to.32bit
    (LAMBDA (a.point)
    (PROG (new.point)
        (BREAK1
            (if (LESSP (fetch xcoord of a.point) 0.0)
             then
```

```
                (SETQ new.point
                      (point.sum a.point
                                 (point.new 360.0
                                 90.0)))
              else
                (SETQ new.point
                      (point.sum a.point
                                 (point.new 0.0
                                 90.0))))
            T
            (latlon.to.32bit around if) NIL)
        (RETURN
            (point.truncate
                (point.scaleby new.point factor))))
))
```

20.3.1 Function Breakpoints

BREAK allows you to set a breakpoint for a function. It takes the form

```
Function:       BREAK
# Arguments:    1
Arguments:      1) an expression, EXPRESSION
Value:          A list of the functions which are broken.
```

BREAK is an NLAMBDA, nospread function. EXPRESSION may be either an atom or a list. If EXPRESSION is an atom, BREAK performs

```
(BREAK0 expression T)
```

whereas if it is a list, BREAK performs

```
(APPLY (FUNCTION BREAK0) expression)
```

Note that the structure of the list must correspond to the arguments expected by BREAK0.

A Definition for BREAK
We might define BREAK as follows:

```
(DEFINEQ
   (break
     (NLAMBDA (x)
```

```
            (MAPCONC
                 (COND
                      ((ATOM x) (LIST x))
                      (T x))
                 (FUNCTION break.1))
       )))
```

where BREAK.1 is defined as

```
(DEFINEQ (break.1 (x)
     (COND
          ((OR (ATOM x)
               (EQUAL (CADR x) 'IN))
               (break0 x T))
          (T
               (APPLY 'break0 x)))
     ))
```

The test for the CADR of x picks up the case where we are using the form (⟨fn1⟩ IN ⟨fn2⟩).

20.3.2 Defining a Breakpoint

BREAK0 actually defines a breakpoint for a function. It is invoked by BREAK or TRACE. It takes the form

```
Function:       BREAK0
# Arguments:    3
Arguments:      1) a function, FN
                2) a condition for breaking, WHEN
                3) a command list, COMS
Value:          The name of the function.
```

BREAK0 modifies the definition of FN to call BREAK1 (an example has been given above) with BRKEXP equivalent to the original definition of FN. WHEN, FN, and COMS are substituted for the values of BRKWHEN, BRKFN, and BRKCOMS, respectively.

The original definition of FN is assigned to an atom created by GENSYM (see Section 9.2.1). The name of this atom is stored on FN's property list under the property BROKEN. The definition of the breakpoint is stored on FN's property list under the property BRKINFO. It has the value (BREAK0 ⟨when⟩ ⟨coms⟩). The purpose for storing this information is to allow you to rebreak a function at a later time. Finally, FN is added to the list BROKENFNS.

```
←(BREAK point.sum)
(point.sum)

←(GETPROP 'point.sum 'broken)
(point.sumb0010)

←(GETPROP 'point.sum 'brkinfo)
((BREAK0 T NIL))

←(PP point.sumb0010)
(point.sumb0010
     (LAMBDA (point1 point2)
          (point.new
               (PLUS (fetch xcoord of point1)
                     (fetch xcoord of point2))
               (PLUS (fetch ycoord of point1)
                     (fetch ycoord of point2)))
     ))

←brokenfns
(point.sum)
```

FN may take several different forms. BREAK0 processes them as follows:

1. If FN is not defined, BREAK0 displays the message (⟨function⟩ not a function).

   ```
   ←(BREAK0 'point.divide)
   ((point.divide not a function))
   ```

2. If FN is a list, it may be a list of functions. BREAK0 is invoked for each member of the list. The values of WHEN and COMS are used for each function.

   If you want to set breakpoints for all the functions in a file, you might use the invocation

   ```
   (BREAK0 (FILEFNSLST ⟨filename⟩) ⟨when⟩ ⟨commands⟩)
   ```

3. FN may have the form (⟨fn1⟩ IN ⟨fn2⟩). This form allows you to set a breakpoint on a function, that is called from many functions, within a specific function.

   BREAK0 changes the name of ⟨fn1⟩ to that of a new function, ⟨fn1⟩-IN-⟨fn2⟩, wherever it appears in ⟨fn2⟩. It substitutes ⟨fn1⟩-IN-⟨fn2⟩ for ⟨fn1⟩ in the expression FN.

   Note: This form works like BREAKIN (see Section 20.3.4), but works on compiled functions whereas BREAKIN works only on interpreted functions.

Consider the following example:

```
←(BREAK '((point.sum point.difference point.quotient
point.times) IN (rectangle.expandby rectangle.center
rectangle.extent)) (NOT (NULL point1))(EVAL (point1
point2) ?= OK))
(point.sum-in-rectangle.expandby point.sum-in-
rectangle.center (point.sum not found in
rectangle.extent) point.difference-in-
rectangle.expandby ...)

←brokenfns
(point.quotient-in-rectangle.center point.difference-
in-rectangle.extent ...)

←(PP rectangle.expandby)
(rectangle.expandby
      (LAMBDA (a.rectangle a.point)
           (rectangle.new
                (point.difference-in-rectangle.expandby
                     (fetch origin of a.rectangle)
                     a.point)
                (point.sum-in-rectangle.expandby
                     (fetch corner of a.rectangle)
                     a.point)))
      ))

←(GETPROP 'point.sum-in-rectangle.expandby 'brkinfo)
((BREAK0 (NOT (NULL point1)) (EVAL (PRINT point1) ?=
OK))
```

A Definition for BREAK0

We might define BREAK0 as follows:

```
(DEFINEQ (break0 (fn when coms)
   (PROG (fn-def brk.condition temp value)
   top
     (SETQ value fn)
     (COND
          ((NULL (ATOM fn))
               (RETURN (break0.subfunction)))
          ((NULL (SETQ fn-def (GETD fn)))
               (AND
                  (SETQ fn-def (FNCHECK fn T))
                  (SETQ fn fn-def)
                  (GO top))
```

```
        (PUTPROP fn 'broken (GENSYM))
        (*
             If the function is not defined, set
             up the breakpoint anyway.
        )
        (putd fn
              (LIST 'nlambda
                    (GENSYM)
                    (LIST 'break1
                          NIL
                          when
                          fn
                          coms)))
        (SETQ value
              (CONS fn '(not a function)))
        (GO set-brokenfns))
((AND (EXPRP fn-def)
      (LISTP (CADDR fn-def))
      (EQUAL (CAADDR fn-def) 'break1))
      (*
          If the function is already broken,
          modify the breakpoint.
      )
      (RPLACA
         (SETQ brk.condition (get.when fn-def))
               when)
         (RPLACD
               (CDR brk.condition) (LIST coms))
         (COND
            ((SETQ temp
                 (ASSOC 'break0
                        (GETPROP fn 'brkinfo)))
                 (RPLACD temp (LIST when coms))
                 (RPLACA 'brokenfns
                      (CONS fn
                            (DREMOVE
                            brokenfns)))
                 (RETURN (LIST fn)))
              (T
                 (SETQ fn-def
                    (CONS (CAR fn-def)
                       (CONS (CADR fn-def)
                             (CDADR
                               (CADDR fn-
                               def)))))
```

```
                    )))
    (SETQ fn-def (SAVED fn 'broken) fn-def))
    (putd fn
          (LIST (CAR fn-def)
                (CADR fn-def)
                (LIST 'break1
                      (CONS 'PROGN (CDDR fn-def))
                      when
                      fn
                      coms)))
  set-brokenfns
    (COND
         ((NULL (MEMBER fn brokenfns))
                (RPLACA brokenfns
                        (CONS fn brokenfns))))
    (ADDPROP fn 'brkinfo (LIST 'break0 when coms))
    (RETURN
         (LIST value)))
    ))

(DEFINEQ (get-when (expression)
    (CDDR (CADDR expression))
    ))

(DEFINEQ (break0-subfunction nil
    (COND
       ((NEQ (CADR fn) 'IN)
             (MAPCONC fn
                      (FUNCTION break0-subfunction.1))))
    ))

(DEFINEQ (break0-subfunction.1 (x)
    (break0 x when coms))
    ))
```

20.3.3 Activating a Breakpoint

A breakpoint is activated by calling **BREAK1** which has been placed in a function definition by BREAK0 or BREAKIN. It takes the form

```
Function:       BREAK1

# Arguments:    6

Arguments:      1) a break expression, BRKEXP
                2) a condition for breaking, BRKWHEN
                3) an expression to be evaluated, BRKFN
```

```
                4) a list of breakpoint commands,
                   BRKCOMS
                5) the type of breakpoint, BRKTYPE
                6) an error indication, ERRORN

Value:          Determined by the Break Package commands.
```

BREAK1 is an NLAMBDA function.

BRKWHEN determines whether or not a break is to occur. Usually, BRKWHEN is an expression that tests some condition of the variables or arguments used by the function to be broken. If BRKWHEN evaluates to NIL, BRKEXP is evaluated and its value is returned as the value of BREAK1.

```
←(BREAKIN sum.list (AFTER GO))
searching...
SUM.LIST

←(EDITF sum.list)
Note: you are editing a broken function
EDIT
*PP
(LAMBDA (x)
     (PROG (y sum)
          (SETQ y (COPY x))
          (SETQ sum 0)
     loop
          (COND
               ((NULL y)
                    (RETURN sum)))
          (SETQ sum (PLUS sum (CAR y)))
          (SETQ y (CDR y))
          (GO loop))
     (BREAK1 NIL T (SUM.LIST after GO) NIL))
*OK
(SUM.LIST)
```

where BRKWHEN has the value (AFTER GO).

When a break occurs, ERRORN is inspected. If it is a list whose CAR is a number, ERRORMESS (see Section 18.6.1) is called to print an identifying message. If its CAR is not a number, ERRORMESS1 is called to print the message. Otherwise, no preliminary message is printed. Normally, you do not define ERRORN. After this, the message (⟨brkfn⟩ broken) is printed.

The Break Package executes the commands that it finds in BRKCOMS, if any. If none of these commands exits the break, the Break Package prints its prompt (e.g., :) and accepts commands from the terminal, and interprets them. You may only leave a breakpoint using the commands GO, !GO, OK, !OK,

RETURN, and ↑ (up arrow). EVAL evaluates BRKEXP and stores its value in the variable !VALUE, whence it may be inspected. You may define other commands via BREAKMACROS.

BRKTYPE specifies the type of break. It takes the values

NIL for user-specified breakpoints
INTERRUPT for CTRL-H breakpoints
ERRORX for error breakpoints

20.3.4 Breaking into a Function

BREAK allows you to set a breakpoint for an entire function. Sometimes, you may want to break within a function, particularly one which is lengthy or complex. To do so, you can use **BREAKIN** which inserts a call to BREAK1 before, after, or around a choice expression within a function.

A breakpoint is inserted by a sequence of editor commands. Alternatively, you may enter the Editor from BREAKIN, search for the proper location yourself, and exit, whence the breakpoint is inserted before, after, or around the current expression.

BREAKIN takes the form

```
Function:       BREAKIN

# Arguments:    4

Arguments:      1) a function name, FN
                2) a location, WHERE
                3) a condition for breaking, WHEN
                4) a command list, COMS

Value:          The name of the function.
```

BREAKIN is an NLAMBDA function.

FN must be a function which is modifiable (i.e., an interpreted function). BREAKIN returns (⟨name⟩ not a function) if the function is undefined or (⟨name⟩ unbreakable) if it is compiled.

When BREAKIN is called on an interpreted function, it displays the message "searching ..." while it calls the Editor to look for the current expression matching the WHERE specification. WHERE specifies the location at which the call to BREAK1 will be inserted as modified by WHEN.

```
(DEFINE
   (convert.latlon.to.32bit (latlon)
      (PROG (new.point)
          (if
               ((LESSP latlon:xcoord 0.0)
```

```
                then (SETQ new.point
                           (point.sum latlon
                                      (point.new 360.0
                                      90.0))))
                else (SETQ new.point
                           (point.sum latlon
                                      (point.new 0.0
                                      90.0)))))
          (RETURN
               (point.truncate
                    (point.scaleby new.point factor))))
   ))
```

Let us break this function around the if...then...else expression:

```
←(BREAKIN convert.latlon.to.32bit (AROUND if) T)
searching...
CONVERT.LATLON.TO.32BIT
←(EDITF convert.latlon.to.32bit)
Note: you are editing a broken function
EDIT
*PP
(LAMBDA (latlon)
   (PROG (new.point)
     (BREAK1
        (if
          ((LESSP latlon:xcoord 0.0)
          then (SETQ new.point
                     (point.sum latlon
                                (point.new 360.0 90.0))))
     else (SETQ new.point
                     (point.sum latlon
                                (point.new 0.0 90.0)))))
        T (around if) NIL)
     (RETURN
          (point.truncate
               (point.scaleby new.point factor))))
))
*OK
(CONVERT.LATLON.TO.32BIT)
```

When a function is broken into, BREAKIN adds the property BROKEN-IN to the function names' property list with the value T. It also adds the property

BRKINFO to the property list with the value (⟨where⟩ ⟨when⟩ ⟨coms⟩). FN is added to the list BROKENFNS.

```
←(GETPROP 'convert.latlon.to.32bit 'BROKEN-IN)
T

←(GETPROP 'convert.latlon.to.32bit 'BRKINFO)
(((AROUND if) T NIL))
```

You may insert multiple breakpoints in a function by setting WHERE to a list of expressions each of which specifies a location at which the function is to be broken.

```
←(BREAKIN convert.latlon.to.32bit
          ((BEFORE PROG) (AFTER point.scaleby)
           (AROUND if) (AFTER RETURN))
          (NOT (NULL latlon)))
searching...
searching...break inserted AFTER (RETURN (POINT.TRUNCATE
&))
searching...
searching...
CONVERT.LATLON.TO.32BIT

←(GETPROP 'convert.latlon.to.32bit 'BRKINFO)
(((BEFORE PROG) (NOT (NULL latlon)) NIL) ((AFTER
point.scaleby) (NOT (NULL latlon)) NIL) ((AROUND if) (NOT
(NULL latlon)) NIL) ((AFTER RETURN) (NOT (NULL latlon))
NIL))
```

Note that WHEN is used for each breakpoint.

If you do not know where you want to insert a breakpoint, you may specify the form (BEFORE TTY:) or (AFTER TTY:) as the value of WHERE. BREAKIN invokes the Editor. You may then search for the proper location at which to insert the breakpoint. When you exit from the Editor via OK, BREAKIN inserts the call to BREAK1 before, after, or around the current expression. To exit without inserting a breakpoint, type the command STOP to TTY:. It acts like an unsuccessful edit command in the original form. The Editor aborts. BREAKIN returns (not found) after the "searching..." message.

BREAKIN prevents you from inserting breakpoints inside an expression that would never be activated. The variable NOBREAKS is a list of atoms. Initially, its value is (GO QUOTE *). If you attempt to insert a breakpoint inside an expression *beginning* with one of these atoms, BREAKIN will not permit it. You may add to NOBREAKS additional atoms that would begin expressions you do not wish to be broken into. However, you should exercise caution in selecting the atoms to be added.

Consider the following example:

```
←(PP SUM.LIST)
(LAMBDA (x)
     (PROG (y sum)
          (SETQ y (COPY x))
          (SETQ sum 0)
     loop
          (COND
               ((NULL y) (RETURN sum)))
          (SETQ sum (PLUS sum (CAR y)))
          (SETQ y (CDR y))
          (GO loop)))

←(BREAKIN sum.list (AFTER GO))
searching...
SUM.LIST

←(EDITF sum.list)
Note: you are editing a broken function
EDIT
*PP
(LAMBDA (x)
     (PROG (y sum)
          (SETQ y (COPY x))
          (SETQ sum 0)
     loop
          (COND
               ((NULL y)
                    (RETURN sum)))
          (SETQ sum (PLUS sum (CAR y)))
          (SETQ y (CDR y))
          (GO loop))
     (BREAK1 NIL T (SUM.LIST after GO) NIL))
*OK
(SUM.LIST)
```

For BEFORE or AFTER commands, BREAKIN ensures that the breakpoint is inserted in a "safe" place:

```
←(BREAKIN sum.list (AFTER TTY:))
EDIT
*3
*7
*2
*OK
```

```
break inserted AFTER (SETQ SUM (PLUS SUM &))
SUM.LIST
```

The three numbers above are Editor commands to descend into the structure of SUM.LIST. In fact, they make the current expression be SUM in the (SETQ SUM ...) expression. BREAKIN does not allow you to insert a breakpoint in the middle of an expression, so it places it "safely" after the expression.

20.3.5 BREAKCHECK: When to Break

When an error occurs, INTERLISP must decide what action to take: either unwind the stack or go into a break. If a computation has proceeded only a little way, such as hitting a misspelled function name at type-in, INTERLISP usually wants to unwind the stack, print an error message, and have you try again. In the midst of a complex computation, the best tactic is to go into a break. The decision on entering a break depends on the depth (e.g., number of function calls) of the computation and the amount of time invested in it. **BREAKCHECK** is called to determine if a break should occur. It takes the form

```
Function:       BREAKCHECK

# Arguments:    2

Arguments:      1) an error position, ERRORPOS
                2) an error number, ERXN

Value:          T, if a break should occur; otherwise,
                NIL.
```

BREAKCHECK is called when an error is encountered. It must decide whether or not to induce a break.

ERRORPOS is the stack position where the error occurred, i.e., it is a stack pointer to the frame representing the function in which the error occurred. ERXN is an error number.

BREAKCHECK returns T to indicate that a break should occur if the depth of the stack is greater than HELPDEPTH. HELPDEPTH is initially set to 7, but you may reset it.

If the stack length (in frames) is less than HELPDEPTH, BREAKCHECK calculates the time spent in the computation. If the time is greater than HELPTIME (in milliseconds), BREAKCHECK will return T to indicate that a break should occur. HELPTIME is initially set to 1000 (milliseconds).

BREAKCHECK searches backwards in the stack from ERRORPOS looking for an ERRORSET frame. If the stack is to be unwound, INTERLISP will do so back to the ERRORSET frame. At the same time, it counts the number of invocations of EVAL. When the number of calls to EVAL exceeds HELPDEPTH, BREAKCHECK returns T. Otherwise, it searches until an ERRORSET frame is found or the top of the stack is reached. It then counts the number

of function calls between the ERRORSET frame (or the top of the stack). The number of functions calls plus the number of calls to EVAL is used as the computation depth. If it exceeds HELPDEPTH, BREAKCHECK returns T.

The computation time is determined by subtracting the value of HELPCLOCK from the value of a call to CLOCK, which reads the system clock. HELPCLOCK is the last recorded value of a call to CLOCK for each call to LISPX. If this exceeds HELPTIME, BREAKCHECK returns T. You may disable the time citerion for breaking by setting HELPTIME to NIL or a very large number.

You may disable all error breaks by resetting the variable HELPFLG to NIL using SETTOPVAL (e.g., you must set the top-level value). HELPFLG is rebound during calls to LISPX (e.g., in a stack frame so assigning a value via SETQ just affects the most recent binding). You may force a break to occur on every error by rebinding the top-level value of HELPFLG to BREAK!.

20.4 UNBREAKING FUNCTIONS

You may unbreak functions in two ways: either issuing the UB command from within the break or executing a function. Functions to unbreak functions are described in this section.

20.4.1 Unbreaking a Function: 1

UNBREAK removes a breakpoint from one or more functions. Its takes the form

```
Function:       UNBREAK
# Arguments:    1
Argument:       1) an expression, EXPRESSION
Value:          A list of functions that are unbroken.
```

UNBREAK is an NLAMBDA, nospread function.

EXPRESSION may be atomic or a list of atoms that are function names. Each function is unbroken via a call to UNBREAK0. Its value is the list of values returned by UNBREAK0.

UNBREAK will unbreak functions that have been broken by BREAK, BREAK0, TRACE, or BREAKIN.

If EXPRESSION is NIL, UNBREAK will unbreak all functions on BROKENFNS. BRKINFOLST is first set to NIL:

```
←(UNBREAK)
(point.sum point.difference point.quotient)
```

If EXPRESSION has the value T, UNBREAK unbreaks the first function on BROKENFNS. Since BREAK, BREAK0, BREAKIN, or TRACE always

add a newly broken function to the front of the list, the effect is to unbreak the most recently broken function.

```
←(BREAK point.sum)
(point.sum)

←(BREAK point.difference)
(point.difference)

←BROKENFNS
(point.difference point.sum)

←(UNBREAK T)
(point.difference)
```

A Definition for UNBREAK

We might define UNBREAK as follows:

```
(DEFINEQ unbreak
     (NLAMBDA x
          (COND
               ((EQUAL (CAR x) T)
                    (SETQ x (LIST (CAAR brokenfns))))
               ((NULL x)
                    (SETQ x (REVERSE brokenfns))
                         (RPLACA 'brokenfns NIL)
                         (RPLACA 'brkinfolst NIL)))
          (MAPCONC x (FUNCTION unbreak0))
     )))
```

20.4.2 Unbreaking a Function: 2

The workhorse for unbreaking functions is **UNBREAK0**. It restores a function to its original state. UNBREAK0 takes the form

```
Function:       UNBREAK0

# Arguments:    1

Arguments:      1) a function name, FN

Value:          The value of FN.
```

FN is treated as follows:

1. If FN is atomic, and was not broken, UNBREAK0 displays the message (⟨fn⟩ not broken) and does nothing.

```
←(UNBREAKO 'point.sum)
(point.sum not broken)
```

2. If FN is atomic and was modified by BREAKIN, UNBREAKIN is called to remove these modifications.
3. If FN is atomic, it is restored to its original state.
4. If FN has the value ⟨fn1⟩-IN-⟨fn2⟩, UNBREAK0 restores ⟨fn2⟩ to its original state. All dummy functions created as a result of the breakpoint are eliminated.
5. If FN is a list of the form (⟨fn1⟩ IN ⟨fn2⟩), UNBREAK0 operates on the functions ⟨fn1⟩-IN-⟨fn2⟩ as in case (4) above.

The value of UNBREAK0 is the name of the function that is restored.

20.4.3 Unbreaking a Broken-into Function

UNBREAKIN allows you to remove a breakpoint that was inserted into a function by BREAKIN. It performs the proper editing actions to remove the breakpoint and restore the function to its original state. It is automatically called by UNBREAK if a function has the property BROKEN-IN. It takes the form

```
Function:       UNBREAKIN
# Arguments:    1
Argument:       1) a function name, FN
Value:          The value of FN.
```

20.5 BREAK PACKAGE UTILITIES

Several functions are useful for writing breakmacros and modifying functions. They are gathered here as utility functions.

20.5.1 Reading Break Package Commands

As discussed in Section 20.2.11, you may create your own breakpoint commands as macros that are executed by functions that you specify. When you give a breakmacro as an element of BRKCOMS, you must be able to read any arguments that are associated with the command. **BREAKREAD** allows you to read elements from BRKCOMS. It takes the form

```
Function:       BREAKREAD
# Arguments:    1
```

```
Arguments:      1) A type specification, TYPE
Value:          The next elements in BRKCOMS or the input.
```

If BRKCOMS is non-NIL for any Break Package function, BREAKREAD returns the next command in the list. And it sets the BRKCOMS list to its CDR.

If TYPE is LINE and the break command was typed in at the terminal, BREAKREAD returns the rest of the input buffer (up to a ⟨CR⟩).

A Definition for BREAKREAD

We might define BREAKREAD as follows:

```
(DEFINEQ
    (breakread (flag)
        (COND
            (brkcoms
                (PROG1
                    (CAR brkcoms)
                    (SETQ brkcoms (CDR brkcoms))))
            ((EQUAL flag 'LINE)
                (READLINE))
            (T
                (CAR (READLINE)))
    ))
```

Note that we use READLINE rather than READ when a macro expects some input, because READLINE will wait for you to type something in while READ will return NIL.

20.5.2 Changing Names in Functions

CHANGENAME changes all occurrences of a name FROM to a name TO in a function FN. It is primarily used by break functions to handle the form (⟨fn1⟩ IN ⟨fn2⟩). It takes the form

```
Function:       CHANGENAME
# Arguments:    3
Arguments:      1) a function, FN
                2) a from name, FROM
                3) a to name, TO
Value:          FN if FROM is found; otherwise, NIL.
```

CHANGENAME will change any names within FN, not just function names. It does not perform any modifications on property lists.

20.5.3 Restoring a Virgin Definition

VIRGINFN knows how to restore the original state of a function regardless of the number of times that it has been advised, broken, broken-in, or traced. It takes the form

```
Function:       VIRGINFN
# Arguments:    2
Arguments:      1) a function name, FN
                2) a flag, FLAG
Value:          The original definition of the function.
```

VIRGINFN is used by a number of system functions to restore a function to its original state including PRETTYPRINT, DEFINE, and the compiler. FLAG determines whether the function is modified or a copy is made. PRETTYPRINT calls VIRGINFN with FLAG equal to NIL so that a copy is made, i.e., if you want to prettyprint a function without any of the encumbering detail that is added by advising, breaking, or tracing the function. If FLAG is T, as specified by the compiler and DEFINE, VIRGINFN physically restores the function to its original state. It also tells you what changes it has made, e.g., one of ⟨FN⟩ UNBROKEN, ⟨FN⟩ UNADVISED, or ⟨FN⟩ NAMES RESTORED.

20.5.4 Printing a Backtrace of the Stack

BAKTRACE implements the BT, BTV, BTV+, BTV*, and BTV! commands of the Break Package. It takes the form

```
Function:       BAKTRACE
# Arguments:    4
Arguments:      1) an initial stack position, IPOS
                2) an ending stack position, EPOS
                3) a skipping function, SKIPFN
                4) options, FLAGS
Value:          A display of the stack from IPOS to EPOS
                subject to the actions of SKIPFN and the
                values of FLAGS.
```

IPOS and EPOS are stack frame specifications as described in Section 30.3.

SKIPFN is a function that determines whether or not to print the information for a stack frame. If SKIPFN is non-NIL, it is applied to the value of (STKNAME (POS)). If the value is T, the current position is skipped (i.e., not printed).

FLAGS specifies the display options for the backtrace. It takes the values for the corresponding commands:

Value	Command
0	BT
1	BTV
5	BTV+
7	BTV*
47Q	BTV!

When BAKTRACE prints information about a stack frame, it will abbreviate information about various sequences of function calls by a single entry. The function calls to be abbreviated are given by BAKTRACELST. Each entry on BAKTRACELST is a list of the form

```
(⟨framename⟩ ⟨key⟩ . ⟨pattern⟩)
```

or

```
(⟨framename⟩ (⟨key1⟩ . ⟨pattern1⟩)
             ...
             (⟨keyN⟩ . ⟨patternN⟩))
```

where a ⟨pattern⟩ is a list of elements that have either atoms, which match a single frame, or lists, which are interpreted as a list of alternative patterns.

For the Fugue Release of INTERLISP-D, BAKTRACELST has the following format:

```
((APPLY
    (**BREAK**
        LISPX ERRORSET BREAK1A BREAK1)
    (**TOP**
        LISPX ERRORSET EVALQT T)
    (**EDITOR**
        LISPX ERRORSET ERRORSET ERRORSET EDITL1
        EDITL0 ERRORSET
        ((ERRORSET ERRORSET ERRORSET EDITL1 EDITL0
        ERRORSET) -)
        EDITL ERRORSET ERRORSET EDITE ((EDITF)
                                       (EDITV)
                                       (EDITP)
                                       -))
    (**USEREXEC**
        LISPX ERRORSET ERRORSET USEREXEC))
```

```
(EVAL
    (**BREAK**
         LISPX ERRORSET BREAK1A BREAK1)
    (**TOP**
         LISPX ERRORSET EVALQT T)
    (**EDITOR**
         LISPX ERRORSET ERRORSET ERRORSET EDITL1
         EDITL0 ERRORSET
         ((ERRORSET ERRORSET ERRORSET EDITL1 EDITL0
         ERRORSET) -)
         EDITL ERRORSET ERRORSET EDITE ((EDITF)
                                        (EDITV)
                                        (EDITP)
                                        -))
    (**USEREXEC**
         ERRORSET LISPX ERRORSET ERRORSET USEREXEC))
(PROGN
    **BREAK**
         EVAL
         ((ERRORSET BREAK1A ERRORSET BREAK1)
          (BREAK1))
(BLKAPPLY
    **BREAK**
         PROGN EVAL ERRORSET BREAK1A ERRORSET BREAK1)
(*PROG*LAM
    (NIL EVALA *ENV*)
    (NIL CLISPBREAK1)))
```

BAKTRACE scans the stack frame by frame. At each frame, it compares the current frame name with the frame names that appear on BAKTRACELST using ASSOC. If the current frame name appears, it attempts to match a contiguous set of stack frames with each of the pattern(s). If a match is unsuccessful, BAKTRACE prints the corresponding key, and continues from the point where the match was successful. If the frame does not appear or the match fails, BAKTRACE simply prints the frame name, unless SKIPFN applied to the frame name is non-NIL. It then proceeds to the next higher frame.

In constructing patterns, you may use the following features:

1. The atom & can be used to match any frame, i.e., it is equivalent to "don't care."
2. The atom—(e.g., a dash) can be used to match nothing, i.e., it can be used for optional matches.

3. You do not have to provide matches for dummy frames, i.e, frames for which DUMMYFRAMEP (see Section 30.5.1), is true. When matching, the matcher automatically skips over these frames when they do not match.
4. If the match succeeds and the key is NIL, nothing is printed (see the last expression for *PROG*LAM above).

21

Advising

Advising is a means of altering the interface to a function. Alteration may occur before or after the function is executed. As a user, you do not need to know how a function works (usually) to modify its interface with its environment. BREAK, TRACE, and BREAKDOWN all use advising to effect the operation of functions.

You may advise functions so that they are affected every time they are called, or when they are called within or by another function. The IRM notes that you may advise TIME as follows:

```
←(ADVISE 'TIME
         'BEFORE
         '(SETQQ U FOO)
         '(PRIN1 PRINT SPACES))
TIME
```

This advice alters TIME so that it prints its results to the file FOO rather than to the terminal.

21.1 ADVISE: MODIFYING A FUNCTION'S INTERFACE

You may advise a function or functions by executing **ADVISE**, which takes the following format

```
Function:       ADVISE
# Arguments:    4
Arguments:      1) a specification, FNS
                2) the time of alteration, WHEN
```

```
                3) the location of alteration, WHERE
                4) the advice, WHAT
Value:          The name of the function(s) advised.
```

FNS is a specification that describes the function(s) to be altered. It may take one of several forms:

1. If FNS is an atom, it is a single function to be advised.
2. If FNS takes the form

   ```
   (fns.1 IN fns.2)
   ```

   then FNS.1 is altered throughout FNS.2. If either or both of FNS.1 and FNS.2 are lists of functions, the names are totally distributed across both lists.
3. If FNS is being advised for the first time, then ADVISE modifies the functions as follows:

   a. It creates a new symbol using GENSYM.
   b. The new symbol is given the original definition of FNS.
   c. The name of the new symbol is placed on the property list of FNS under the property ADVISED.
   d. An appropriate S-expression version of FNS is created for FNS.
   e. FNS is added to the list ADVISEDFNS.
   f. The list (WHEN WHERE WHAT) is stored under the property ADVICE on the property list of FNS.

4. If FNS is a list of functions, each function is advised in turn with copies of the other three arguments.

A function may be advised when it is broken. If so, it is unbroken before it is advised. If a function has been advised before (because the ADVISED property exists), it is merely moved to the head of ADVISEDFNS.

If FNS is not defined, ADVISE displays an error message "NOT A FUNCTION" and breaks.

WHEN may take the values BEFORE, AFTER, or AROUND. If it has the value BEFORE or AFTER, the advice is inserted into the function's definition before or after the location specified by WHERE. If WHEN has the value AROUND, then the advice has a * within its form and the body of the function is inserted in place of the *.

WHERE specifies the location of the advice in the function definition. You may place advice around the entire body of a function or specific functions within the function. We may advise READ to print its value each time it is executed using the following expression:

```
←(ADVISE 'READ 'AFTER '(PRIN1 !VALUE))
READ
```

WHERE may take the values LAST, BOTTOM, END, or NIL. If so, the advice is added following all other advice, if any. If WHERE has the value TOP OR FIRST, the advice will be inserted as the first piece of advice. Thus, a function may be subject to multiple pieces of advice both before and after its execution.

What does an advised function look like? If we execute the expression

```
(ADVISE 'point.sum
        'T
        '(BEFORE PLUS)
         '(PRIN1 "EXECUTING PLUS"))
```

we obtain the following definition for POINT.SUM:

```
←(EDITF 'point.sum)
Note: you are editing an advised function
EDIT
*PP
(LAMBDA (point1 point2)
   (ADV-PROG (!value)
      (ADV-SETQ !value
         (ADV-PROG NIL
            (ADV-RETURN
               (PROGN
                  (point.new
                        (PLUS point1:xcoord
                        point2:xcoord)
                        (PLUS point1:ycoord
                        point2:ycoord))))))
   (ADV-RETURN !value)))
```

A Definition for ADVISE

We might define ADVISE as follows:

```
(DEFINEQ
 (advise (fns when where what)
    (PROG (fns.1 fns.2 fns.def)
```

```
top
      (COND
        ((ATOM fns)
           (SETQ fns (FNCHECK fns))
        ((EQUAL (CADR fns) 'IN)
           (SETQ fns.2 (CADDR fns))
           (RETURN
              (COND
                ((ATOM (SETQ fns.1 (CAR fns)))
                   (COND
                     ((ATOM fns.2)
                        (advise1 fns.1 fns.2))
                     (T
                        (MAPCAR fns.2 (FUNCTION
                        advise2))))
                ((ATOM fns.2)
                     (MAPCAR fns.1 (FUNCTION
                     advise3)))
                (T
                     (MAPCONC fns.1 (FUNCTION
                     advise4))))))
        (T
            (RETURN
                     (MAPCAR fns (FUNCTION
                     advise5)))))
        (COND
          ((OR what (NULL when)) NIL)
          ((NULL where)
               (*
                    Advise[⟨function⟩ ⟨advice⟩]
                    is the same as
                    advise[⟨function⟩ BEFORE NIL
                    ⟨advise⟩]
               )
               (SETQ what when)
               (SETQ when 'BEFORE))
           (T
               (SETQ what where)
               (SETQ where NIL)))
        (RESTORE fns 'BROKEN)
        (SETQ fns.def (GETD fns))
        (COND
          ((NULL fns.def)
               (PRIN1 fns)
               (SPACES 1)
```

```
            (PRIN1 "not defined.")
            (TERPRI))
        ((OR (NULL (EXPRP fns.def))
            (NULL (GETP fns 'ADVISED)))
            (SETQ fns.2 (SAVED fns 'ADVISED
            fns.def))
            (PUTD fns
                  (LIST (CAR fns.2)
                        (CADR fns.2)
                        (SETQ fns.2
                              (SUBPAIR 'DEF
                              (LIST
                                (COND
                                  ((CDR
                                    (SETQ fns.2
                                          (CDDR
                                          fns.2)))
                                   (CONS 'PROGN
                                   fns.2))
                                  (T
                                    (CAR
                                    fns.2))))
                              (COPY ADVISE-
                              FORM))))))
           (T
             (SETQ fns.2 (CADDR fns.def))))
        (*
             Moving a function that is already
             advised to the front of the list of
             advised functions.
        )
        (SETATOMVAL 'ADVISEFNS
                    (CONS fns
                          (/DREMOVE fns ADVISEFNS)))
        (SETQ fns.1 when)
loop
        (SELECTQ fns.1
          (NIL
               (*
                    Advise[⟨function⟩] implies
                    setting up for advising and
                    exiting
               )
               (RETURN fns))
          (BEFORE
```

```
                    (SETQ fns.2
                          (CDDR
                              (CADDR
                                   (CADDR fns.2)))))
               (AFTER
                    (SETQ fns.2 (CDDDR fns.2))
               (AROUND
                    (SETQ fns.2
                          (CAR
                              (LAST (CADDR (CADDR
                              fns.2)))))
                    (COND
                         ((NEQ (CAR fns.2) 'ADV-RETURN)
                              (ERROR "ADVISE" when '?)
                              (RETURN)))
                    (RPLACA (CDR fns.2)
                            (SUBPAIR'*
                                    (LIST (CADR fns.2
                                    what))))
                    (GO exit))
               (BIND
                    (NCONC (CADR fns.2)
                           (COND
                              ((ATOM what)
                                   (LIST what))
                              (T
                                   (APPEND what))))
                    (GO exit))
               (ERROR "ADVISE" when '?))
          (COND
               ((NULL where)
                    (ATTACH what (LAST fns.2)))
               (T
                    (ADD.ADVICE fns.2 what where T)))
     exit
          (ADDPROP fns 'ADVICE (LIST when where what))
          (RETURN fns))
     ))
```

where the subfunctions are defined as follows:

```
(DEFINEQ
     (advise1 (fns.1 fns.2 a.flag)
          (PROG (fns.3)
               (COND
```

```
                    ((NOT (ATOM fns.3))
                        (RETURN fns.3))
                    (a.flag
                        (advise fns.3 (COPY when)
                                      (COPY where)
                                      (COPY what)))
                    (T
                        (advise fns.3 when where what)))
                (RETURN fns.3))
    ))

(DEFINEQ
    (advise2 (a.fns.2)
        (advise fns.1 a.fns.2 T)
    ))

(DEFINEQ
    (advise3 (a.fns)
        (advise1 a.fns when T)
    ))

(DEFINEQ
    (advise4 (a.fns)
        (MAPCAR fns.2 (FUNCTION advise41))
    ))

(DEFINEQ
    (advise41 (a.fns.2)
        (advise1 a.fns a.fns.2 T)
    ))

(DEFINEQ
    (advise5 (a.fns)
        (advise a.fns (COPY when) (COPY where) (COPY
        what))
    ))
```

21.2 UNADVISE: REMOVING ADVICE

You may remove advice from a function by executing UNADVISE, which takes the form

Function:	UNADVISE
# Arguments:	1
Arguments:	1) an expression, EXPRESSION
Value:	A list of the functions that have been unadvised.

UNADVISE is an NLAMBDA, nospread function. For each function found in EXPRESSION, UNADVISE removes the properties created by ADVISE as well as any program code added to the body of the function that was specified by ADVISE.

UNADVISE will save on the list ADVINFOLST enough information to allow you to READVISE a function at a later time.

UNADVISE, when executed with no arguments, unadvises all functions that are described on ADVISEDFNS. It sets ADVINFOLST to NIL.

UNADVISE, when executed with an argument of T, unadvises the first function on ADVISEDFNS. That is, it unadvises the last (most recent) function that was advised.

21.3 READVISING A FUNCTION

You may restore a function to its advised state by executing **READVISE**. READVISE uses information found on ADVINFOLST or under the property READVICE to reinstate the advice for the specified functions. It takes the form

```
Function:       READVISE
# Arguments:    1
Arguments:      1) an expression, EXPRESSION
Value:          A list of functions that have been
                readvised.
```

READVISE is an NLAMBDA, nospread function. If you readvise a function, the advice will be stored under the property READVICE on the atom's property list. Thus, you may advise a function once, then unadvise and readvise it many times without having to specify all of the original information.

READVISE, when executed with no arguments, will readvise every function that is found on ADVINFOLST. Thus, you can turn advising on and off for a large number of functions with a simple function invocation.

21.4 SAVING ADVICE IN A FILE

As noted in Section 17.2.10, the File Package provides a file package command for saving advice for functions between sessions. PRETTYDEF creates the proper entries in a file to readvise the functions saved in the file when it is reloaded. It uses **ADVISEDUMP** to place the proper information in the file. ADVISEDUMP takes the form

```
Function:       ADVISEDUMP

# Arguments:    2

Arguments:      1) a function, FNS
                2) a flag, FLAG

Value:          A list of the functions dumped.
```

If FLAG is T, ADVISEDUMP writes two expressions to the primary output file: a DEFLIST expression and a READVISE expression. However, if FLAG is NIL, only the DEFLIST expression is written to the file. The difference is that the former corresponds to the action of readvising a function while the latter corresponds to the process of advising. In either case, the advice is copied to the property READVICE of the function which makes it "permanent" since all properties and their values will also be dumped to the file.

A Definition of ADVISEDUMP

We might define ADVISEDUMP as follows:

```
(DEFINEQ
    (advisedump (fns flag)
        (*
            FLAG is T for READVISE and NIL for ADVISE.
        )
        (SETQ fns
            (MAPCONC fns
                    (FUNCTION advise.dump.1)))
        (MAKEDEFLIST fns 'READVICE)
        (COND
            (flag
                (PRINTDEF (CONS 'READVISE fns))))
    ))

(DEFINEQ
    (advise.dump.1 (fn)
        (MAPCAR (PACK-IN- fn)
                (FUNCTION advise.dump.2))
    ))

(DEFINEQ
  (advise.dump.2 (fn)
    (PROG (property-list)
          (SETQ property-list (GETPROP fn 'ADVICE))
          (COND
            (property-list
```

```
                    (PUTPROP fn
                             'READVICE
                             (CONS (GETPROP fn 'ALIAS)
                                   (APPEND property-
                                   list)))))
            (RETURN fn))
))
```

22

DWIM: Automatic Error Correction

DWIM means *Do-What-I-Mean*. It is an automatic error correction facility that assists you in correcting errors that do not affect the structure or logic of your program. Many of the errors made by INTERLISP programmers are simple ones that are easily recognized and fixed, so much so that another program could watch what you type in or inspect a program as it is read from a file and correct these errors. DWIM is a program that performs this function although (as the IRM notes) it only operates on unbound atoms and undefined functions at the present time. When an error occurs, DWIM inspects the current context of the computation, attempts to figure out what is wrong and how it may be corrected, attempts to correct it (possibly with confirmation), and lets the computation proceed as if no error had occurred.

22.1 DWIM MODES

DWIM can operate in one of three modes: disabled, cautious, or trusting.

When DWIM is *disabled*, it will not interfere with your activities. DWIM may be disabled by executing (DWIM NIL) or setting DWIMFLG to NIL.

In *cautious* mode, DWIM determines what correction is to be applied, if any, and then seeks approval from you. You may answer YES or NO to any suggested correction. DWIM uses an internal interval timer to determine how long to wait for a user response. If the interval expires, DWIM proceeds automatically with the correction. DWIM may be placed in cautious mode by executing (DWIM 'C). This is also the initial setting of DWIM when it is first loaded.

DWIM may also be placed in *trusting* mode. In this mode, DWIM performs the appropriate corrections without requiring you to approve them. DWIM may be placed in trusting mode by executing (DWIM 'T).

When DWIM makes a correction, it is contained only within the current image of the program, e.g., the virtual memory. To preserve the correction, the user will have to execute MAKEFILE to update the disk-based version of the program.

22.1.1 A DWIM Example

Consider a function that you have entered from a file via LOAD.

```
(*   COPYLIST-makes a copy of its argument   *)
(DEFINEQ
(copylist (x)
     (COND
          ((NULL x) NIL)
          ((LIST-P x)
               (APEND (LIST (copylist (CAR x)))
                      (copylst (CDR x))))
          (T x))
))
```

Let us assume that DWIM currently operates in cautious mode. We can execute COPYLIST as follows:

```
←(SETQ x '(a b))
(A B)
←(SETQ y (copylist x))
```

There are obvious errors of spelling in the definition of COPYLIST. DWIM detects these errors and responds:

```
LIST-P [in COPYLIST] → LISTP? ...Yes
APEND [in COPYLIST] → APPEND? ...Yes
COPYLST [in COPYLIST] → COPYLIST? ...Yes
(A B)
```

In this case, the default answer is YES and the corrections have been applied. Once all errors are corrected, INTERLISP calculates the value of the function.

In trusting mode, when a function is already defined via LOAD or the editor, DWIM operates as in cautious mode. However, when expressions are interactively entered via LISPX (see Section 25.2), DWIM makes the correction and tells you what it did. If you were to type in

```
←(SEQT fruits (NCOCN 'apples 'oranges))
=SETQ
=NCONC
ORANGES
```

would be the response from INTERLISP where DWIM interceded to correct the spellings of SETQ and NCONC.

22.2 DWIM PROTOCOLS

A protocol is an interchange between DWIM and the user concerning actions to be taken to correct errors. Currently, DWIM corrects three types of errors:

1. Spelling errors
2. Parenthesis Typing Errors
3. T Clause Errors

22.2.1 Spelling Correction

Whenever INTERLISP encounters a function or an atom it does not recognize, it invokes DWIM (if DWIMFLG is not NIL) to see if spelling correction is required. If the error did not occur in type-in mode, DWIM displays the following message (see example above):

```
⟨name⟩ [in ⟨function-name⟩] → ⟨corrected name⟩?
```

DWIM proceeds based on the value of APPROVEFLG. If APPROVEFLG is NIL, DWIM makes the correction and continues. Otherwise, it waits for your response. This response takes one of the following forms:

Y whence DWIM echoes **es** and makes the correction.

N whence DWIM echoes **o** and does not make the correction.

↑ whence DWIM does not make the correction and does not cause an error.

CNTRL-E whence DWIM prints U.D.F. and forces a break.

Do nothing, whence DWIM waits a specified interval and then types ... followed by the default answer (e.g., the value of FIXSPELLDEFAULT).

⟨SP⟩ or ⟨CR⟩ whence DWIM waits indefinitely.

Note that when you type N, you are instructing DWIM's spelling corrector to return NIL. This lets the function decide what to do next.

Modifying the Wait Interval

The interval for which DWIM will wait for your response is specified by DWIMWAIT. Initially, the value of DWIMWAIT is 10 seconds. You may reset the value of DWIMWAIT to meet your own requirements. Setting DWIMWAIT to 100 provides you (roughly) one and a half minutes to think about your answer. Setting DWIMWAIT to an arbitrarily large number (⟩ 10000) should provide you enough time to mull over the problem and formulate a response. Setting DWIMWAIT to 0 causes DWIM to take the default immediately.

Modifying the Spelling Default

After the wait interval has expired and you have not responded, DWIM will substitute the default answer and continue. The default answer is specified by FIXSPELLDEFAULT. Its initial value is Y(es). You may reset it via

```
←(SETQ FIXSPELLDEFAULT 'N)
N
```

22.2.2 Parenthesis Errors

A typical error that results when you are entering a large program is to substitute a "9" for a "(' or a "0" for a ")". This results from failure to push the SHIFT key (on the Xerox 1100 keyboard). DWIM recognizes when such errors occur and corrects them in a manner similar to spelling correction.

```
←(SETQ states 9LIST 'maryland 'virginia 'newyork))
= ( LIST
(maryland virginia newyork)
```

where DWIM has detected the parenthesis error and corrected it.

Unfortunately, "(" and ")" often occur over other keys than "9" or "0" on other keyboards. There is no mechanism for specifying the relationship between the parentheses (left or right) and the erroneous keys. Thus, DWIM fails to work in some cases for foreign terminals.

When DWIM attempts to correct a parenthesis error, it waits for 3*DWIMWAIT seconds. This allows you time to consider the expression and, perhaps, back up over the last parenthesis and continue typing.

22.2.3 Clause Errors

A *clause error* occurs when a (T ...) clause is misplaced within a conditional expression. If uncorrected, INTERLISP treats the T clause as a function invocation, does not recognize T as a function, and responds with a U.D.F. error. DWIM recognizes and corrects three cases of this error:

1. The T clause appears outside the conditional expression. A previous clause is entered with an additional right parenthesis which completes the conditional expression.

```
(COND
     ((ATOM x) ...)
     ((LISTP x) ...)
     ((STRINGP x) ...))    ← extraneous parenthesis
     (T ...))
```

   where the ... means we don't care what follows.

2. The T clause appears within a previous conditional clause. A right parenthesis is omitted from a previous conditional clause.

```
(COND
     ((ATOM x) ...)
     ((LISTP x) ...)
     ((STRINGP x) ... &     ← missing parenthesis
                  (T ...))
```

3. A T clause is surrounded by an additional pair of parentheses. Generally, this is a typing error resulting from a psychological mistake, e.g. following a pattern for creating conditional clauses in the expression.

```
(COND
     ((ATOM x) ...)
     ((LISTP x) ...)
     ((STRINGP x) ...)
     ((T ...)))
```

Any other type of U.D.F. error is not currently recognized by DWIM, so the error will cause a break.

Fixing the Error

DWIM automatically corrects a T clause error on type-in. It prints T FIXED and continues.

If APPROVEFLG is NIL, DWIM corrects the error and prints a message consisting of [IN ⟨function name⟩] followed by one of the incorrect COND forms and →. On the following line, DWIM prints the correct COND form and proceeds.

```
←(DEFINEQ
    (FACT (n)
         (COND
              ((ZEROP n) 1))
              (T
                   (ITIMES n
                           (FACT (SUB1 n)))))
    ))
(FACT)

←(SETQ APPROVEFLG NIL)
NIL

←(FACT 3)
[IN FACT] (COND ...) (T ...) ->
          (COND ... (T ...))
6
```

If APPROVEFLG is T, DWIM prints U.D.F. T followed by [IN ⟨function name⟩], a few spaces, and FIX?. DWIM waits for approval (unless DWIM-

WAIT is exceeded). An answer of Y allows DWIM to proceed as if APPROVEFLG had a value of NIL.

```
←(SETQ APPROVEFLG 'T)
T
←(FACT 3)
U.D.F. T [IN FACT]   FIX? Y      "typed by the user"
(COND ... (T ...))
CONTINUE WITH T CLAUSE? Y
6
```

Proceeding with the Computation

Once DWIM makes a correction, it must decide how to proceed with the computation. Each of three cases is treated separately.

When the T clause is outside the COND expression (case 1), DWIM cannot know whether the last clause of the COND succeeded or not. That is, if the T clause were enclosed within the COND expression, would it have been executed? Thus, DWIM will ask you whether or not it should continue with the T clause (see example above). The default answer is YES. If you answer NO, DWIM proceeds to execute the expression following the (previously) erroneous T clause.

When the T clause is embedded in the previous clause (case 2), DWIM moves the T clause to its proper place in the COND expression. The T clause was reached through successful execution of & (see case above). The value of & must be returned as the value of the COND expression, but this value is no longer around. Therefore, DWIM asks you "OK TO REEVALUATE" followed by the expression corresponding to &. If you respond Y, DWIM reevaluates & and proceeds. Otherwise, DWIM aborts the computation, prints U.D.F. T, and causes an error.

Before requesting your approval, DWIM determines whether or not it is safe to reevaluate the form. It will do so without commenting if the form is atomic or if the CAR of each of its sublists appears on OKREEVALST and each of the arguments can safely be reevaluated. The initial value of OKREEVALST is

```
←OKREEVALST
(AND OR PROGN SAVESETQ CAR CDR ADD1 SUB1 CONS LIST EQ
EQUAL PRINT PRIN1 APPEND NEQ NOT NULL)
```

When the T clause is surrounded by an extra pair of parentheses, DWIM merely removes them and proceeds with the computation.

22.3 ERROR CORRECTION ALGORITHMS

Whenever INTERLISP encounters an S-expression whose first atomic form has no binding (that is, no current value), it invokes FAULTEVAL. If DWIM is enabled, DWIMBLOCK is called by FAULTEVAL to attempt to treat the error.

DWIMBLOCK determines what correction is to be made, if any, and displays it for you. If you disapprove of the correction by entering N, CNTRL-E, or ↑, DWIMBLOCK returns NIL and allows FAULTEVAL to handle the error. If DWIM is allowed to correct the error, it returns the corrected value (via RETEVAL—see Section 30.8.3) to FAULTEVAL. In this case, FAULTEVAL assumes the returned value is an erroneous form and attempts to execute it. However, since the form is now corrected, it will execute properly and the computation will proceed.

DWIM can correct three types of errors: unbound atoms, undefined functions (represented by the CARs of functional forms), and undefined functions which are the arguments of APPLY. Algorithms for correcting these three error types are discussed in the next three sections.

22.3.1 Unbound Atoms

An unbound atom is one that has not been assigned a value (even though the value stored in its value cell is NOBIND). This error usually occurs when an atom name is used in an expression without previously being used in a SETQ or is defined in a PROG without being assigned a value.

DWIM uses the following algorithm to attempt to correct an unbound atom:

1. If the first character of the atomic form is ', DWIM assumes you intended to specify the form '⟨atom name⟩. DWIM substitutes (QUOTE ⟨atom name⟩) and proceeds. If ' is followed by an S-expression, then DWIM assumes that you wanted the entire S-expression quoted. In either case, no message is displayed and your approval is not sought. If just the single character ' appears, DWIM gives up.

   However, the interpretation of ' on type-in is controlled by a read table (usually) and so '⟨atom name⟩ will automatically be converted to (QUOTE ⟨atom name⟩). You may view how this operates by typing in the definition of a new function and then applying PP to it. Each '⟨atom name⟩ will be replaced by (QUOTE ⟨atom name⟩).

```
←(DEFINEQ
    (test.for.stop (argument)
        (EQUAL argument 'STOP)
    ))
(TEST.FOR.STOP)

←(PP TEST.FOR.STOP)
(TEST.FOR.STOP
    (LAMBDA (argument)
        (EQUAL argument (QUOTE STOP))))
```

2. If CLISP is enabled (see Chapter 23), DWIM determines if the atom is part of a CLISP construct. If so, the appropriate transformation is applied and returned.
3. If an atom contains a "9", DWIM assumes that you intended it to be a "(". It calls the editor to repair the expression containing the atom. DWIM attempts to repair the entire expression by balancing parentheses.

   Note that a form like

   ```
   (LIST x9CAR y) becomes (LIST x (CAR y))
   ```

   where the parentheses have been balanced after the 9 has been replaced by a (.

   Note that this feature does not work if "(" is the shift character for another key. For example, on my Lear-Siegler ADM-3A, the "(" is the shift character for "8". There is no way to tell DWIM how to treat these kinds of terminals.
4. A similar action takes place if an atom contains a "0" except that the replacement is a ")".

   Note that other terminals may have the ")" as the shift character for the "9". There is no way to tell DWIM how to handle this case.
5. If an atom begins with a 7 (on TTY-type keyboards), DWIM assumes you intended to type ' and performs the replacement.

   ```
   ←(SETQ super.bowl.XVIII 7los.angeles.raiders)
   los.angeles.raiders

   ←super.bowl.XVIII
   los.angeles.raiders
   ```

   Note that certain keyboards, like the ADDS Viewpoint, have the " as the SHIFT value of the ' key. DWIM assumes that a " is a valid demarcator for a string in this case and cannot correct an atom error.
6. If the atom is a valid Editor command, DWIM assumes that you intended to call the Editor. It does so (by invoking EDITF), and executes the command. At this point, you are in the Editor and must follow its conventions. Valid edit commands are stored on the Editor variable EDITCOMSA.
7. DWIM looks on DWIMUSERFORMS for S-expressions. It evaluates each one in turn. If a non-NIL value is returned, this value is returned by DWIMBLOCK as the corrected value of the atom. DWIMUSERFORMS allows you to tailor DWIM's error correction algorithm to specific users. Words which are difficult to spell might be intercepted by expressions on DWIMUSERFORMS and corrected. In general, it will

be quicker to correct a word through expressions on DWIMUSERFORMS than through the spelling corrector. Section 22.7.1 describes the parameters to be used in writing these expressions.

8. If the unbound atom occurs in a function, DWIM attempts to correct the spelling of the atom using the LAMBDA and PROG variables (if any) of the function. Here DWIM assumes that the atom may be a valid variable whose name is misspelled.
9. If the unbound atom was entered in response to a broken function, DWIM attempts to correct spelling using the LAMBDA and PROG variables of the broken function.
10. Finally, DWIM will attempt to correct the spelling of the atom name using SPELLINGS3 (see Section 22.7.2).
11. Otherwise, DWIM fails.

22.3.2 Undefined Functions

Properly, this error is known as an "undefined CAR of form." However, this error typically occurs when an atom appears as the first atom of an S-expression without a corresponding function definition.

DWIM uses the following algorithm to attempt to correct this error:

1. If the ATOM is T, DWIM assumes a misplaced T clause and attempts to correct the conditional (see Section 22.2.3).
2. If the atom is F/L, DWIM replaces F/L with FUNCTION (LAMBDA and proceeds.

   Note F/L is a shorthand notation for specifying the form "FUNCTION (LAMBDA". DWIM will look for a variable list by determining how many S-expressions follow the F/L. If only one appears, DWIM substitutes (x) as the variable list.

```
←(F/L (ADD1 x))
(LAMBDA (X) (ADD1 x))
```

   because only one S-expression follows F/L.

   F/L is used only as a DWIM construct. Because it limits the portability of your code, we do not recommend its use.
3. If the atom is IF or another CLISP statement operator, the appropriate CLISP transformation is perfomed and the result returned to FAULTEVAL.
4. If the atom has a function definition, DWIM attempts spelling correction on the CAR of the definition using the value of LAMBDASPLST. LAMBDASPLST initially has the value (LAMBDA NLAMBDA).

5. If the atom has an EXPR or CODE property, DWIM prints ⟨atom name⟩ UNSAVED, executes UNSAVEDEF to restore the definition to the function cell, and continues. No approval is requested.
6. If the atom has the property FILEDEF, its definition is to be found in a file. DWIM looks for the definition. If it is found, DWIM asks you if it should be loaded. If you approve, the definition is loaded and execution proceeds.

```
←(LOADFROM 'COMPLEX)
⟨KAISLER⟩COMPLEX..4

←(PUTPROP 'CMULT 'FILEDEF 'COMPLEX)
COMPLEX

←(DWIM 'T)
TRUSTING

←(LOADFNS 'COMPLEX 'COMPLEX)
(COMPLEX)

←(SETQ CX1 (COMPLEX 3.0 4.0))
loading from ⟨KAISLER⟩COMPLEX..4
PRINT.COMPLEX
loading from ⟨KAISLER⟩COMPLEX..4
REAL
loading from ⟨KAISLER⟩COMPLEX..4
IMAG
((3.0 . 4.0))

←(CMULT CX1 CX2)
loading from ⟨KAISLER⟩COMPLEX..4
CMULT
UNBOUND ATOM
CX2
```

7. If the atom is a CLISP construct and CLISP is enabled, the appropriate transformation is applied and execution continues.
8. If the atom contains an 9, DWIM assumes a (was intended, performs the substitution, and continues.
9. Similarly, if the atom contains a 0, DWIM assumes a) was intended, performs the substitution, and continues.
10. If the CAR of the form is a list, DWIM attempts to correct the spelling of the CAAR of the form (e.g., the CAR of the list) using LAMBDASPLST. If successful, DWIM returns the corrected expression.

```
((LAMBDA (X) (SELECTQ X (...))) ...)
```

may be a form that selects and returns a function name to be applied to the remaining arguments. If the LAMBDA is misspelled, using LAMBDASPLST, DWIM corrects it.

```
←((LAMBD (x) (ADD1 x)))
=LAMBDA
NON-NUMERIC ARG
NIL
```

11. If the error occurred in typein mode, and the CAR is a small number, DWIM assumes it is an edit command, invokes the Editor (via EDITF), and executes the command.

```
←(3)
=CMULT                          'last function loaded'
edit
*
```

12. If the CAR of the form is an Editor command, DWIM invokes the editor (via EDITF) and executes the command.

```
←(R X Y)
=CMULT
edit
X ?                    'deliberate error for illustration'
*
```

Note that the result of applying either case 11 or 12 is that the user remains within the Editor and is subject to its conventions.

13. DWIM evaluates the expressions on DWIMUSERFORMS. If any S-expression returns a non-NIL value, this value is treated as the corrected form, it is evaluated and its value returned by DWIM.
14. DWIM attempts spelling correction using SPELLINGS2 (see Section 22.7.2).
15. Otherwise, DWIM fails.

22.3.3 Undefined Functions in APPLY

An undefined function occurring as an argument to APPLY is treated in a manner similar to an undefined CAR of form. The major difference is that DWIM will also attempt spelling correction using SPELLINGS1.

22.4 ENABLING DWIM

DWIM enables or disables automatic error correction. It takes the following form

```
Function:       DWIM
# Arguments:    1
Argument:       1) a mode, MODE
Value:          The old mode.
```

MODE may take one of the following values to specify how DWIM will operate:

1. If MODE is NIL, automatic error correction is disabled.
2. If MODE is C, DWIM is enabled in cautious mode.

```
←(DWIM 'C)
CAUTIOUS
```

3. If MODE is T, DWIM is enabled in trusting mode.

```
←(DWIM 'T)
TRUSTING
```

4. All other values cause an error.

```
←(DWIM 'X)
not on DWIMODELST.
```

DWIMODELST is a list of the modes that DWIM recognizes as valid:

```
←DWIMODELST
((C CAUTIOUS (APPROVEFLG . T)) (T TRUSTING
(APPROVEFLG)))
```

During type-in mode, DWIM always acts as if it were in TRUSTING mode when expressions are submitted for execution. Errors involving 8-9 correction or spelling correction are always treated as if DWIM were in CAUTIOUS mode.

22.5 DWIMIFYING AN EXPRESSION

DWIMIFY acts as a preprocessor for CLISP. It scans an expression given to it as though it were being interpreted. For each expression that would generate an

error, it calls DWIM to fix it. You are consulted (for your approval) when something in the expression is to be corrected. If DWIM is unable to make the correction, no message is printed, and no changes are made.

DWIMIFY performs all corrections and transformations to its first argument that would actually occur if it were run. It takes the form

```
Function:       DWIMIFY

# Arguments:    3

Arguments:      1) a form or function name, EXPRESSION
                2) a quiet flag, QUIETFLG
                3) a list, LST

Value:          Either the result or nothing.
```

EXPRESSION is an expression to be DWIMIFYed. If EXPRESSION is an atom and LST is NIL, then DWIMIFY assumes that EXPRESSION is the name of a function and DWIMIFYs its definition. The result is printed unless QUIETFLG is T.

If EXPRESSION is a list or LST is non-NIL, EXPRESSION is an expression to be DWIMIFYed. If LST is not NIL, it provides an edit push-down list leading to EXPRESSION that describes the context of EXPRESSION so that variable bindings may be determined. This form is primarily used for invoking DWIMIFY from the Editor, the Break Package, and the Programmer's Assistant.

Consider the following examples:

```
←(DWIMIFY '(LAMBD (X) (ADD3 X)))
=LAMBDA
(LAMBDA (X)
     (ADD3 X))
(LAMBDA (X) (ADD3 X))

←(DWIMIFY '(CONS (QUOTE SEAN CONNERY)))
(possible) parenthesis error in
(QUOTE SEAN CONNERY)
too many arguments (more than 1)
(CONS (QUOTE SEAN CONNERY))
```

When an attempt to correct a function or variable fails, the name is added to a corresponding internal list. On subsequent encounters, DWIMIFY will not attempt to operate on names appearing on either of these lists. These lists are initialized to NOFIXFNSLST and NOFIXVARLST respectively.

DWIMIFY never attempts corrections on global variables in order to avoid affecting the operation of other portions of your program. Similarly, no attempt will be made to correct variables which are defined:

as LOCALFREEVARS in block declarations
as SPECVARS in block declarations
via DECLARE expressions in the function body

In order to work properly, DWIMIFY needs a lot of information about how the interpreter works. Among the "things" that it knows are

The systax of various expressions
NLAMBDA arguments are not evaluated
How variables are to be bound
When to correct or not correct the spelling of variables occurring in the expression
How to correct errors for CAR of expressions
When an expression has too many arguments, which are symptomatic of parenthesis errors
If PROG labels contain CLISP characters

Note: DWIM has not evolved very much from its initial development. This is unfortunate because the concept is a powerful one. Do not expect DWIM to operate correctly in every case; it operates unevenly at best.

22.5.1 DWIMIFYing a List of Functions

An alternative form of DWIMIFY, **DWIMIFYFNS**, DWIMIFYes each function on its argument list. It takes the form

Function:	DWIMIFYFNS
# Arguments:	1
Argument:	1) A list of functions, FNSLST
Value:	The list of functions that have been DWIMIFYed.

DWIMIFYFNS is an NLAMBDA, nospread function. If the CAR of FNSLST is atomic, it is assumed to be the name of a file whose functions are to be DWIMIFYed. If so, DWIMIFYFNS invokes FILEFNSLST (see Section 17.7.3) on the CAR of FNSLST. The file must already be loaded for DWIMIFYFNS to operate properly.

```
←(LOADFROM 'COMPLEX)
⟨KAISLER⟩COMPLEX..4

←(DWIMIFYFNS COMPLEX)
CPLUS not defined.
```

```
←(LOAD 'COMPLEX)
⟨KAISLER⟩COMPLEX..4
FILE CREATED 17-Oct-84 21:20:25
COMPLEXCOMS
⟨KAISLER⟩COMPLEX..4

←(DWIMIFYFNS COMPLEX)
[in ROUNDTO] (possible) parenthesis error in
(ROUNDED (QUOTIENT X Y) Y)
too many arguments (more than 1)
(COMPLEX REAL IMAG CPLUS CDIFFERENCE CZERO CMULT
PRINT.COMPLEX FLOOR ROUNDTO SIGN ROUNDED RECIPROCAL
TRUNCATE PRINT.ARRAY FACTORIAL)
```

DWIM discovered an error in ROUNDTO because ROUNDED expects only one argument, as evidenced by the following display:

```
←(PP ROUNDED)
(ROUNDED
     (LAMBDA (X) **COMMENT**
          (TRUNCATE (PLUS X (QUOTIENT (SIGN X)
                                      2.0)))))
```

which we correct via the Editor. Thus, DWIMIFYing functions in a file is one way of locating errors.

22.5.2 DWIMIFY Variables

DWIMIFY uses a number of variables to determine how it should proceed with the different situations it encounters in an expression. These variables are

NOFIXVARSLST	A list of variables that DWIMIFY will not try to correct. You may use this list to prevent DWIMIFY from expending needless effort on correcting variable spellings.
NOFIXFNSLST	A list of functions that DWIMIFY will not try to correct (for the same reason as above).
NOSPELLFLG	If NOSPELLFLG is T, DWIMIFY will not perform any spelling correction. Its initial value is NIL. NOSPELLFLG is reset to T when the compiler is called to

compile functions whose definitions are loaded from a file. For example,

```
←(SETQ NOSPELLFLG T)
(NOSPELLFLG reset)
T
←(DWIMIFY '(LAMBD (X) (ADD3 X)))
(LAMBD (X)
     (ADD3 X))
```

DWIMIFYCOMPFLG — If T, DWIMIFY is called before an expression is compiled.

DWIMCHECK#ARGSFLG — If T, DWIMIFY will check whether or not the expression, whose CAR is assumed to be a function name, has arguments. Its value is initially T. Consider the example

```
←(SETQ DWIMCHECK#ARGSFLG NIL)
(DWIMCHECK#ARGSFLG reset)
NIL
←(DWIMIFY '(CONS (QUOTE SEAN CONNERY)))
(CONS (QUOTE SEAN CONNERY))
```

where DWIM does not checks the argument count for QUOTE.

DWIMCHECKPROGLABELSFLG — If T, DWIMIFY will check PROG labels for CLISP characters.

DWIMMESSGAG — If T, DWIMIFY does not print any error messages. Its value is initially NIL.

CLISPHELPFLG — If NIL, DWIMIFY will not ask you to approve any CLISP translations that it makes. Rather, it assumes NO where your approval would be required. Its value is initially T.

CLISPRETRANFLG — If T, DWIMIFY retranslates all expressions which have remote

```
translations in the CLISP hash
array. Its value is initially
NIL.
```

22.6 THE SPELLING CORRECTOR

The objective of the spelling corrector is to produce the correct word, called the *respelling*, given a word that is assumed to be incorrect. To do so, it uses a "closeness" measure which is inversely proportional to the number of disagreements between two words—the word to be corrected, XWORD, and a candidate—and directly proportional to the length of the longer word. The spelling corrector examines a spelling list, SPLST, provided by the user, from which it selects candidates. The closeness between XWORD and each candidate is computed. An internal list of closest candidates is generated and retained.

If a word is found on SPLST for which there are no disagreements, the spelling corrrector returns that word as the respelling of XWORD. Alternatively, if it has found no disagreements and only one "closest" word, this word is returned as the respelling.

XWORD may contain one or more ⟨ESC⟩s. ⟨ESC⟩ will match any number of characters in a potential candidate. The entire spelling list is always searched. If more than one respelling is found, the spelling corrector prints "AMBIGUOUS" and returns NIL.

Each respelling, whether approved by you or not, is moved to the front of SPLST. Because many respellings have no disagreements, the time required to correct the spelling of frequently misspelled words is significantly reduced. These words appear at the beginning of the list which is searched from front to back, so they are likely to be found earlier when spelling correction is attempted.

The spelling corrector counts the number of disagreements between two words. This number, when divided by the length of the longer of the two words, gives a measure of the relative disagreement between the two words. This number is always less than or equal to one (where one indicates total disagreement!). One minus this number yields a measure of the relative agreement or closeness of the two words. (In practice, the spelling corrector uses integer values between 0 and 100 in order to avoid the penalties of floating point arithmetic.)

Consider the words MAKEFILE and MKAEFILE. They disagree in positions 3 and 4 so the number of disagreements is 2. The length of both words is 8 characters, so the measure of relative disagreement is 25 (e.g., (2/8)*100). Therefore, the measure of relative agreement is 75.

The spelling corrector usually uses a relative agreement of 70. This allows it to correct single substitution errors in four character words.

22.6.1 Choosing a Candidate

CHOOZ is the central function of the spelling corrector. It takes the form

```
Function:       CHOOZ

# Arguments:    7

Arguments:      1) a word to be corrected, XWORD
                2) a spelling list, SPLST
                3) a minimum relative agreement, REL
                4) a tail flag, TAIL
                5) an optional function, FN
                6) a tie flag, TIEFLAG
                7) number of doubled characters, DOUBLES

Value:          A list of possible candidates for the
                respelling.
```

CHOOZ examines the spelling list one word, TWORD, at a time. TWORD is rejected as a candidate if:

1. If FN returns NIL when applied to TWORD.
2. It is too short or too long to be sufficiently close to XWORD.

If REL is 70 and XWORD is five characters long, then words greater than 8 characters in length will be rejected. Words shorter than XWORD require special consideration because doubled characters (due to keyboard stuttering or too much pressure on the keys) are not counted as disagreements. CHOOZ handles doubled characters in XWORD before scanning SPLST. CHOOZ uses the number of doubled characters to decide whether or not to reject a word based on length.

If TWORD is not rejected, CHOOZ computes the number of disagreements between TWORD and XWORD by calling SKOR (see below).

FN is a function of one argument, TWORD. If FN is NIL, it is assumed equivalent to (LAMBDA NIL T) which means all words on SPLST are considered as candidates. A simple definition for FN might compare its first character with that of XWORD (which it uses freely):

```
(LAMBDA (tword)
    (EQ
        (CAR (UNPACK tword))
        (CAR (UNPACK xword))))
```

A Definition for CHOOZ

We might define CHOOZ as follows:

```
(DEFINEQ
   (chooz (xword rel splst tail fn tieflag doubles)
 (*
```

```
    Convert XWORD to a list of characters.
)
(COND
    ((NLISTP xword)
        (SETQ xword (CHCON xword))))
(PROG (xlength tlength tword twordlst synonym score
value)
      (*
        Get the length of XWORD.
      )
      (SETQ xlength (LENGTH xword))
      (*
        Count the number of doubled characters.
      )
      (AND
        (NULL doubles)
        (SETQ doubles 0)
        (MAPCAR xword (FUNCTION count-doubles)))
loop
      (COND
        ((NULL splst)
            (*
                If no candidates to inspect, do
                nothing.
            )
            (GO exit.chooz))
        ((NULL (SETQ tword (CAR splst)))
            (*
                Get next word on spelling list. Note
                that we check for end of spelling
                list via NULL.
            )
            (SETQ splst (CDR splst))
                (GO loop))
        (T
            (SETQ splst (CDR splst))))
      (COND
        ((LISTP tword)
            (*
                If tword is a list, then the second
                element is a synonym for the first.
            )
            (SETQ synonym (CDR tword))
                (SETQ tword (CAR tword))))
                (SETQ tlength (NCHARS tword))
```

```
(COND
    ((COND
       ((IGREATERP tlength xlength)
        (*
             Check to see if the
             difference in the number of
             characters between tword
             and xword is sufficient to
             make it unnecessary to call
             SKOR; xlength divided by
             tlength must be less than
             rel.
        )
        (ILESSP
           (IQUOTIENT (ITIMES xlength
             100)
                      tlength)
           rel))
       ((AND (NULL tail)
             (ILESSP
                (IQUOTIENT
                   (ITIMES tlength 100)
                   (IDIFFERENCE xlength
                                doubles))
                rel)))
        (*
             If XWORD is longer
             than TWORD, allow for
             the possibility of
             doubled characters.
        )
        (GO loop))
((AND
    (SETQ twordlst (chcon tword))
    (SETQ score
          (SKOR xword twordlst xlength
                tlength))
    (OR
       (NULL fn)
       (COND
         ((EQUAL fn 'GETD)
              (*
                   Get the definition of
                   the comparison
```

```
                    function from the
                    synonym.
                )
                (GETD synonym))
            (T
                (APPLY* fn synonym))))
    (SETQ twordlst
        (COND
            ((LISTP tword)
                (*
                    Distinguish from
                    a runon
                    correction which
                    is returned as a
                    dotted pair.
                )
                (LIST tword synonym))
            (T tword)))
    (COND
       ((LISTP score)
          (AND runonflg
               tail
               (OR
                  (NULL value)
                  (EQUAL tieflag
                         'EVERYTHING)
                  (IGREATERP tlength
                             (NCHARS
                               (CAAR
                             value))))
               (SETQ value
                     (CONS
                        (CONS temp
                              (PACKC
                              score))
                        (COND
                           ((EQ tieflag
                           'EVERYTHING)
                                 value)))))
       ((ZEROP score)
          (COND
             ((EQ tieflag 'EVERYTHING)
                 (SETQ value
                       (CONS temp
                             value)))
```

```
                                ((AND
                                   (NEQ tieflag 'ALL)
                                   (NEQ tieflag 'LIST))
                                   (*
                                      Return the value.
                                   )
                                   (SETQ value temp)
                                   (RETURN value))
                                ((NEQ rel 100)
                                   (*
                                      Tieflag equal to LIST
                                      means list the tied
                                      candidates.
                                   )
                                   (SETQ rel 100)
                                   (SETQ value (LIST temp)))
                                 (T
                                   (SETQ value
                                         (CONS temp
                                         value)))))
                          ((IGREATERP
                              (SETQ score
                                    (chooz1 xlength
                                            tlength
                                            score))
                                    rel)
                              (SETQ value
                                    (CONS temp
                                          (COND
                                             ((EQ tieflag
                                             'EVERYTHING)
                                                 value)
                                             (T
                                                (SETQ rel
                                                score)
                                                NIL)))))
                           ((EQ score rel)
                              (SETQ value
                                    (CONS twordlst
                                                value))))))
    (GO loop)
exit.chooz
    (SETQ value
          (COND
             ((OR
```

```
                    (EQ tieflag 'ALL)
                    (EQ tieflag 'LIST)
                    (EQ tieflag 'EVERYTHING))
                    (COND
                        ((CDR value)
                            (REVERSE value))
                        (T value)))
               ((AND
                    (CDR value)
                    (NULL tieflag))
                    NIL)
               (T
                    (CAR value))))
     (RETURN value))
))
```

COUNT.DOUBLES merely counts the number of double characters in **XWORD**. It may be defined as:

```
(DEFINEQ
     (count.doubles (x)
          (PROG (last.char)
                (SETQ last.char NIL)
                (COND
                    ((EQUAL x last.char)
                        (SETQ doubles (ADD1 doubles)))
                    (T
                        (SETQ last.char x))))
     ))
```

CHOOZ1 computes the relative closeness of XWORD and TWORD as a percentage by dividing the difference of the avergae number of characters and the number of mistakes. It is defined as

```
(DEFINEQ
   (chooz1 (xlen tlen score)
 (PROG (temp)
       (SETQ temp (IPLUS xlen tlen))
       (RETURN
          (IQUOTIENT
               (ITIMES 100
                       (IDIFFERENCE temp
                                    (ITIMES score 2)))
               temp)))
   ))
```

22.6.2 Scoring a Candidate

SKOR evaluates a candidate and computes the number of disagreements. It takes the form

```
Function:       SKOR
# Arguments:    5
Arguments:      1) the word to be corrected, XWORD
                2) a candidate word, TWORD
                3) the length of XWORD, XLEN
                4) the length of TWORD, TLEN
                5) a flag, FLAG
Value:          The number of disagreements or NIL.
```

SKOR scans both TWORD and XWORD from left to right one character code at a time. The list of character codes is prepared by CHOOZ before it calls SKOR. Corresponding characters agree if:

1. They are, in fact, the same character.
2. They appear on the same key of the keyboard; this handles the case of shift errors when typing characters in (see below).
3. The character of XWORD is the lowercase version of the corresponding character of TWORD.

Characters that agree are discarded.

If the first characters of TWORD and XWORD do not agree, SKOR checks to see if either character is one that has already been encountered. In effect, transposition of characters is handled by looking back in the word rather than looking forward. Displacements of characters by two or fewer (e.g., when a character is omitted in one word) positions are treated as a transposition error. Displacements of more than two positions are treated as disagreement errors. In either case, both characters have been accounted for and are discarded.

When Characters Do Not Agree

If the first characters do not agree and neither agrees with previously accounted for characters, SKOR operates as follows:

1. If TWORD has more characters than XWORD remaining, the first character of TWORD is removed and saved, and comparison of TWORD with XWORD continues with the remaining characters.
2. If TWORD has the same or fewer characters, the first character of XWORD is removed and saved, and comparison of TWORD with XWORD continues. However, a check is made to ensure that it is not a

double character typing error, whence the character would be considered accounted for.

Whenever more than two characters in either XWORD or TWORD are unaccounted for, candidate scoring is aborted because the two words are considered to disagree.

Value Returned by SKOR

When SKOR completes the comparison, it returns a value consisting of:

1. The number of unaccounted-for characters,
2. The number of disagreements,
3. The number of transpositions.

However, this number is subject to two qualifications:

1. If both XWORD and TWORD have a character unaccounted-for in the same position, the two characters are counted only once. That is, substitution errors are counted as one disagreement.
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 editor commands (e.g., XRT becomes XTR).

A Definition of SKOR

We might define SKOR as follows:

```
(DEFINEQ
   (skor (xword tword xlength tlength flag)
(PROG (xbuf1 xbuf2 tbuf1 tbuf2 xfirst tfirst
       xchar index transpositions)
      (*
          Initialize SKOR variables.
          INDEX is the current position in the word.
          TRANSPOSITIONS counts the number of transposed
          characters.
      )
      (SETQ index 0)
      (SETQ transpositions 0)
loop
      (*
           Get the first characters of XWORD and TWORD.
      )
      (SETQ xfirst (CAR xword))
      (SETQ tfirst (CAR tword))
```

```
        (COND
            ((NULL xword)
                (COND
                    ((NULL tword)
                        (*
                            If word to be corrected and
                            candidates are both null,
                            just exit.
                        )
                        (GO exit.skor))
                    (T   (GO loop2))))
            ((NULL tword)
                (GO loop1))
            ((OR
                (EQUAL xfirst tfirst)
                (*
                    Test for upper to lower case
                    substitution.
                )
                (AND
                    (EQUAL xfirst (IPLUS tfirst -64))
                    (IGREATERP xfirst 128)
                    (ILESSP xfirst 170))
                (AND
                    (EQUAL tfirst (IPLUS xfirst -64))
                    (IGREATERP tfirst 128)
                    (ILESSP tfirst 170))
                (*
                    Check for user typing L instead of 1;
                    the IRM being difficult to read.
                )
                (AND
                    (EQUAL tfirst 241)
                    (EQUAL xfirst 211)))
                (SETQ xword (CDR xword))
                (sub1var xlength)
                (SETQ tword (CDR tword))
                (sub1var tlength)
                (SETQ xchar xfirst)
                (GO loop)))
    loop1
        (COND
            ((EQUAL xfirst (CAR tbuf2))
                (*
                    A character has been encountered in
                    TWORD before it has been seen in
```

```
                    XWORD. For example, the P in IPRNT
                    compared to the P in PRINT. Note that
                    RPINT is handled as a transposition
                    of PRINT without consulting the
                    buffers.
               )
               (COND
                    ((IGREATERP (LENGTH tbuf2)
                                (IPLUS xlength 2))
                         (add1var index))
                    (T
                         (add1var transpositions)))
               (SETQ tbuf2 NIL)
               (SETQ xword (CDR xword))
               (sub1var xlength)
               (SETQ xchar (CAR tbuf1))
               (GO loop))
          ((EQUAL xfirst (CAR tbuf1))
               (COND
                    ((IGREATERP (length tbuf1)
                                (IPLUS xlength 2))
                         (add1var index))
                    (T   (add1var transpositions)))
               (COND
                    (tbuf2
                         (SETQ tbuf1 tbuf2)
                         (SETQ tbuf2 NIL))
                    (T
                         (SETQ tbuf1 NIL)))
               (SETQ xword (CDR xword))
               (sub1var xlength)
               (SETQ xtemp xfirst)
               (GO loop))
          ((NULL tword)
               (GO loop3)))
loop2
     (COND
          ((EQUAL tfirst (CAR xbuf2))
               (COND
                    ((IGREATERP (LENGTH xbuf2)
                                (IPLUS tlength 2))
                         (add1var index))
                    (T   (add1var transpositions)))
               (SETQ xbuf2 NIL)
               (SETQ tword (CDR tword))
               (sub1var tlength)
```

```
        (GO loop))
   ((EQUAL tfirst (CAR xbuf1))
        (COND
             ((IGREATERP (LENGTH xbuf1)
                         (IPLUS tlength 2))
                  (add1var index))
             (T   (add1var transpositions)))
        (COND
             (xbuf2
                  (SETQ xbuf1 xbuf2)
                  (SETQ xbuf2 NIL))
             (T
                  (SETQ xbuf1 NIL)))
        (SETQ tword (CDR tword))
        (sub1var tlength)
        (GO loop))
   ((AND xword
         (EQUAL xfirst (CADR tword))
         (EQUAL tfirst (CADR xword))
         (NEQ tfirst (CADDR tword)))
         (*
             Distinguish simple transposition from
             the case where comparison has become
             unsynchronized, e.g., MYCIN versus
             MICIN. If the Y is discarded,
             comparing CIN to ICIN looks like a
             transposition of CI (which is wrong)
             whereas it matches CI if the I is
             discarded.
         )
         (SETQ xtemp (CADR xword))
         (SETQ xword (CDDR xword))
         (sub1var xlength)
         (sub1var xlength)
         (addv1ar transpositions)
         (SETQ tword (CDDR tword))
         (sub1var tlength)
         (sub1var tlength)
         (GO loop))
   ((IGREATERP tlength xlength)
        (COND
             ((NULL tbuf1)
                  (SETQ tbuf1 tword))
             ((NULL tbuf2)
                  (SETQ tbuf2 tword))
```

```
                    (T   (RETURN NIL)))
                (SETQ tword (CDR tword))
                (sub1var tlength)
                (GO loop)))
loop3
      (COND
          ((OR
                (EQUAL xfirst xtemp)
                (EQUAL xfirst (CADR xword)))
                (*
                    Remove a character from XWORD. Check
                    for a double character. First phrase
                    checks if character is equal to last
                    character which occurs when last
                    character was correct. Second phrase
                    checks if equal to next character, so
                    throw the current one away.
                )
                (SETQ xword (CDR xword))
                (SETQ xlengthgth (SUB1 xlength))
                (sub1var xlength))
           ((AND tail
                 (NULL tword)
                 (NULL tbuf1)
                 (NULL tbuf2)
                 (NULL xbuf1)
                 (NULL xbuf2)
                 (ZEROP index)
                 (OR
                     (ILESSP (ITIMES transpositions 4)
                             tlength)
                     (NULL (CDR xword))))
                 (*
                     Don't handle runon corrections when
                     there is a transposition unless the
                     word is very long.
                 )
                 (RETURN
                     (COND
                       ((AND flag
                             CLISPFLG
                             (MEMBER (CHARACTER (CAR
                             xword))
                                     CLISPCHARS))
                             (*
```

```
                                        Do not consider runon
                                        corrections when
                                        encountering CLISP
                                        operators. If X*Y appears
                                        in your program, where X is
                                        bound, but Y is not, then
                                        DWIM should not offer X *Y
                                        as a possibility.
                                    )
                                    NIL)
                               (T xword))))
                    (T
                         (COND
                              ((NULL xbuf1)
                                   (SETQ xbuf1 xword))
                              ((NULL xbuf2)
                                   (SETQ xbuf2 xword))
                              (T   (RETURN NIL)))
                         (SETQ xword (CDR xword))
                         (sub1var xlength)
                         (SETQ xtemp xfirst)))
               (GO loop)
exit.skor
               (COND
                    ((AND
                         (NULL xword)
                         (NULL tword)
                         tbuf1
                         xbuf1)
                         (SETQ tbuf1 (LENGTH tbuf1))
                         (SETQ xbuf1 (LENGTH xbuf1))
                         (AND
                              tbuf2
                              (SETQ tbuf2 (LENGTH tbuf2)))
                         (AND xbuf2
                              (SETQ xbuf2 (LENGTH xbuf2)))
                         (COND
                              ((OR
                                   (EQUAL tbuf1 xbuf1)
                                   (EQUAL tbuf1 xbuf2))
                                   (*
                                        Check for substitution
                                        errors. Subtract 1 so when
                                        2 gets added below the net
                                        effect is 1.
                                   )
```

```
                    (sub1var index)))
               (COND
                    ((AND
                         tbuf2
                         (OR
                              (EQUAL tbuf2 xbuf1)
                              (EQUAL tbuf2 xbuf2)))
                         (sub1var index)))))
     (SETQ index
          (IPLUS index
               (COND
                    (xbuf2 2)
                    (xbuf1 1)
                    (T 0))
               (COND
                    (tbuf2 2)
                    (tbuf1 1)
                    (T 0))))
     (RETURN
          (IPLUS index transpositions)))
))
```

22.7 DWIM PARAMETERS

DWIM uses several parameters to determine how it will perform automatic error correction (including spelling correction). These variables are described in the following table:

DWIMFLG	If NIL, suppresses all automatic error correction operations. It is initially NIL. This flag is set when you execute the function DWIM.
FIXSPELLREL	The default value for REL that is used by FIXSPELL when REL is NIL. It is initially 70. You may reset FIXSPELLREL to any value you want. Obviously, lower values will allow greater disagreement between the word to be checked and the candidate words.
FIXSPELLDEFAULT	The default answer when the user does not respond to a request for approval of a spelling correction. Its initial value is Y (for yes). When DWIMIFY is

	invoked, this variable is temporarily rebound to N (for no).
DWIMWAIT	The number of seconds that DWIM waits after requesting approval for some action. If the user does not respond within this time, DWIM assumes the default value (see discussion above).
ADDSPELLFLG	If NIL, this flag suppresses calls to ADDSPELL (see Section 22.8.1). It is initially T.
RUNONFLG	If NIL, this flag suppresses any run-on spelling corrections. It is initially T. A run-on condition occurs when two words are typed without an intervening space. Runons are difficult to detect, but CHOOZ can determine simple cases.
DWIMUSERFORMS	A list that allows you to specify your own corrections or transformations. Its initial value is ((MACROTRAN) (DWIMLOADFNS?)). It is discussed in more detail below.
DWIMLOADFNSFLG	If T, DWIM loads a function from a file noticed by the File Package when it encounters an undefined function in some operation. It is initially T.
APPROVEFLG	If T, DWIM will ask you for your approval of a correction it intends to make to one of your functions. It is initially NIL.
LAMBDASPLST	DWIM uses the value of LAMBDASPLST to correct bad function definitions or LAMBDA expressions. Initially, it has the value (LAMBDA NLAMBDA). You may want to add to LAMBDASPLST if you define new function types via DWIMUSERFORMS.

22.7.1 User-Directed Corrections and Transformations

DWIMUSERFORMS allows you to specify your own corrections and transformations before spelling correction is invoked. DWIMUSERFORMS is a list of

forms that are evaluated in the order in which they appear. If any expression returns a non-NIL value, this value is treated as an expression to be evaluated. It is evaluated and the resulting value is returned as the value of FAULTEVAL or FAULTAPPLY. If all expressions on DWIMUSERFORMS return NIL, DWIM proceeds normally to attempt spelling correction.

In order for an expression on DWIMUSERFORMS to attempt correction of an expression, it needs to have information about the context of the error. The following DWIM variables are made available to external functions (via SPECVARS) so that your functions may inspect them.

FAULTX	Its value is an S-expression for errors concerning unbound atoms or CAR of expressions. Its value is the name of the function for undefined functions in APPLY. When an error occurs, DWIM places the cause of the error in FAULTX for you to inspect.
FAULTARGS	Its value is the list of arguments appearing in an S-expression when an error of undefined function in APPLY occurs. To correct errors in the argument list, you may modify or reset FAULTARGS.
FAULTAPPLYFLG	If an undefined function in APPLY occurs, this flag is set to T to distinguish it from the case of an unbound atom (since FAULTX is atomic in both cases). After an expression has been evaluated on DWIMUSERFORMS and has returned a non-NIL value, this flag determines what to do with the result. If the flag is T, the resulting expression is a function to be applied, whereas, if it is NIL, it is an expression to be evaluated.
TAIL	If the error is an unbound atom, TAIL contains the CAR of the TAIL of the expression in which the unbound atom occurred. You can replace or correct the erroneous atom by executing (RPLACA TAIL ⟨expression⟩)

PARENT	If the error is an unbound atom, PARENT is the expression in which the unbound atom appears. Thus, TAIL is set to (TAIL parent).
TYPE-IN?	This flag is true if the error occurred while you were typing at the keyboard.
FAULTFN	This variable contains the name of the function in which the error occurred. Its value is TYPE-IN when the error occurs at type-in. Its value is EVAL or APPLY when those functions are explicitly invoked in your program.
DWIMIFYFLG	This flag is T if an error was encountered during an attempt to dwimify an expression as opposed to running a program.
EXPR	This variable contains the definition of the value of FAULTFN or the expression given to EVAL when an error occurred.

22.7.2 The Spelling Lists

A *spelling list* is merely a list of atoms. Many subsystems define their own spelling lists. DWIM maintains four spelling lists for spelling correction during normal system operation:

USERWORDS
USERWORDS is a list of all words that are defined by the user via SET, SETQ, DEFINEQ, LOAD, and the various editing functions. It is used for spelling correction by a number of different system function including BREAK, EDITF, ADVISE, and PRETTYPRINT. It is initially NIL. LASTWORD always contains the last word that was added to USERWORDS. Thus, if you have just defined a function via type-in and see an error, you may immediately edit it by simply typing (EDITF) or prettyprint it by typing (PP).

SPELLINGS1
SPELLINGS1 is a list of function names used for spelling correction when an input is typed in "apply" format (i.e., using the square bracket protocol) and the function is determined to be undefined. It initially contains the most commonly used system functions. Whenever LISPX (see Section 25.2) is given an input in "apply" format, the name of the function is added to SPELLINGS1.

The initial value of SPELLINGS1 is:

```
('[spellseparator]' DV MAKEFILE DF LOAD FILES?
LOGOUT MOVD ENDLOADUP LOADUP)
```

SPELLINGS2

SPELLINGS2 is a list of function names for all other undefined functions, e.g., those that are usually used in "eval" format. Whenever LISPX is given a non-atomic S-expression, the name of the function in the expression is added to SPELLINGS2. Function names for your functions are also added to SPELLINGS2 by DEFINEQ, LOAD (when loading compiled code), UNSAVEDEF, EDITF, and PRETTYPRINT.

The initial value of SPELLINGS2 is:

```
←SPELLINGS2
('[spellseparator]' CALLS XPRESS SETBACKSPACE
INITBACKSPACE INITFONTS DISPLAYTERMTYPE DISPLAYTERMP
GETECHOMODE GETCONTROL GETRAISE GETDELETECONTROL
HVREADEND HVREADER HPRINTO HREAD HVFWDREAD HVRPTREAD
HVBAKREAD HVFWDCDREAD HPINITRDTBL READVARS
MAKEHVPRETTYCOMS HPRINT1 HPRINT HPRINTBLOCK
HPRINTBLOCKA0009 HPRINTBLOCKA0008 HCOPYALL1 HCOPYALLA0013
HCOPYALL EQUALALL EQUALALLA0005 EQUALALLA0004 COPYALL
COPYALLA0010 BRKDWNCLEAR RESULTS BRKDWNFBOXES BRKDWNINIT
BRKDWNRESULTS BREAKDOWN BRKDWNBOXES BRKDWNTIME
ASSEMBLETRAN READASCIZ SYSOUTP HELPBINSEARCH DODIRECTORY
UNDELFILE /DELFILE /UNDELFILE DODIR DIRECTORY EQMEMBHASH
STORETABLE MSLISTSET UPDATEFN UPDATECHANGED1
UPDATECHANGED UNSAVEFNS)
```

SPELLINGS3

SPELLINGS3 is a list of words used for correcting the spelling of unbound atoms. Whenever LISPX is given an atom to evaluate, the name of the atom is added to SPELLINGS3. Atoms may also be added to SPELLINGS3 if they are edited via EDITV or set via RPAQ/RPAQQ.

When a file is loaded, all variables initialized as the result of File Package commands will also be added to SPELLINGS3.

Atoms are also added to SPELLINGS3 when they are defined by SET or SETQ (as well as to USERWORDS).

The initial value of SPELLINGS3 is:

```
←SPELLINGS3
('[spellseparator]' MSANALYZECOMS SPELLINGS3 SPELLINGS2
INITIALS AFTERSYSOUTFORMS LISPXCOMS X9FontsFamily
FONTDEFS INITCOMS HOSTNAME SYSTEMTYPE CPUTIMEO
CONSOLETIMEO CONSOLETIME CPUTIME EDITTIME GAINSPACEFORMS
```

```
HPRINT.SCRATCH ORIGECHOCONTROL ORIGTERMSYNTAX
ORIGDELETECONTROL DONTCOPYDATATYPES HPRPTSTRING
HPRINTRDTBL HPRINTHASHARRAY HPRINTMACROS HPRINTCOMS
BRKDWNLST BRKDWNLABELS BRKDWNTYPE BRKDWNFLTFMT
BRKDWNTYPES BRKDWNARGS BRKDWNCOMPFLG BRKDWNLENGTH
BRKDWNCOMS NOSWAPFNS JSYSES FILEINFOTYPES DIRCOMMANDS
DIRCOMS UTILITYCOMS HISTORYCOMS MASTERSCOPEDATE
MSNEEDUNSAVE CHECKUNSAVEFLG RECORDCHANGEFN MSBLIP
MSPATHSCOMS MCHECKBLOCKSCOMS DATABASECOMS NODUMPRELATIONS
MSDATABASEINIT MSDBEMPTY MSDBCOMS MSHELPFILE MSAUXCOMS
MSPRINTCNT MSPRINTFLG MSOPENFILES MSFILELST)
```

Spelling List Sections

Each spelling list is divided into two sections separated by a special marker, ⟨spellseparator⟩, which are known as the *permanent* and *temporary* sections. When a new word is first encountered, it is added to the front of the temporary section of the appropriate list. If the word already exists in the temporary section, it is merely moved to the front of that section. If the word is found in the permanent section, no action is taken.

The lengths of the temporary sections for the lists SPELLINGx (x = 1,2,3) are determined by the variables #SPELLINGSx (x = 1,2,3). They are initialized to 30. #USERWORDS determines the length of the temporary section of USERWORDS. It is initialized to 60. You may redefine the values of these variables via SETQ. A large value for #USERWORDS ensures that important words will be recorded, but at the expense of lengthier search times.

When the length of a temporary section exceeds one of the variables described above, the last (e.g., oldest) word is forgotten by deleting it from the temporary section. This mechanism prevents the spelling lists from becoming cluttered with words that may have been used only once as an intermediate variable.

When a word is corrected by the spelling corrector, that word is moved to the front of the corresponding spelling list. Thus, it is moved into the permanent section. Words that have been misspelled, and subsequently corrected, are considered important and will not be forgotten.

22.8 SPELLING FUNCTIONS

Many of the spelling functions are available to you independently of their use by DWIM. You may want to use them when writing your own expressions to be placed in DWIMUSERFORMS or elsewhere in your program.

22.8.1 Adding a Word to a Spelling List

ADDSPELL adds a word to one of the four spelling lists. It takes the form

```
Function:       ADDSPELL
# Arguments:    3
Arguments:      1) a word, WORD
                2) a spelling list index or the list
                   itself, SPLST
                3) an index, INDEX
Value:          The new word added to the list.
```

SPLST is interpreted as follows:

1. If SPLST is NIL, then WORD is added to both USERWORDS and to SPELLINGS2. This value is used by DEFINEQ.
2. If SPLST is 0, then WORD is added to USERWORDS. This value is used by LOAD (see Section 17.9.1) when loading EXPRs to property lists.
3. If SPLST is 1, then WORD is added to SPELLINGS1. This value is used by LISPX (see Section 25.2).
4. If SPLST is 2, then WORD is added to SPELLINS2. This value is used by LISPX (see Section 25.2).
5. If SPLST is 3, then WORD is added to USERWORDS and SPELLINGS3.
6. If SPLST is a spelling list (ie., a list resulting from the evaluation of an argument to ADDSPELL), then INDEX is interpreted as the length of the temporary section. WORD is added to the end of the temporary section.

ADDSPELL sets LASTWORD to the value of WORD when SPLST is NIL, 0 or 3.

If WORD is not a literal atom, ADDSPELL does nothing. If WORD is already a member of a spelling list in its temporary section, WORD is moved to the front of that section.

You may disable ADDSPELL by setting ADDSPELLFLG to NIL.

A Definition for ADDSPELL

We might define ADDSPELL as follows:

```
(DEFINEQ
   (addspell1 (word splst index)
 (AND
     (LITATOM word)
     (SELECTQ splst
          ((NIL 0)
```

```
        (*
             DEFINE specifies NIL for SPLST which
             causes a word to be added to
             SPELLINGS2 because some user function
             must call it. However, the function
             may not be a top level function, so
             do not add it to SPELLINGS1 until it
             is used as such.
        )
        (SETQ userwords
              (addspell1 word userwords
             #userwords))
        (AND
              (NULL splst)
              (SETQ spellings2
                    (addspell1 word
                               spellings2
             #spellings2)))
        (SETQ lastword word))
(1
        (*
             ADDSPELL is called from LISPX for
             APPLY mode inputs; the word is added
             to the permanent section.
        )
        (SETQ spellings1
              (addspell1 word spellings1)))
(2
        (*
             ADDSPELL is called from LISPX for
             EVAL mode inputs; the words are added
             to the permanent section.
        )
        (SETQ spellings2
              (addspell1 word spellings2)))
((T 3)
        (*
             ADDSPELL is called from LISPX for
             variables or by SAVESET, RPAQ, RPAQQ,
             SET, SETQ, or EDITV.
        )
        (SETQ userwords
              (addspell1 word userwords
             #userwords))
        (SETQ spellings3
```

```
                    (addspell1 word
                               spellings3
                        #spellings3))
                (SETQ lastword word))
            (COND
                ((LISTP splst)
                    (addspell1 word splst index))
                (T
                    (ERROR 'Bad addspell type'
                           splst)))))
))
```

DEFINE enters ADDSPELL with SPLST set to NIL. It adds words to SPELLINGS2 because a user function is being defined and some other user function will probably call it. But, it might not be a top level function (e.g., one invoked at typein), so it is not added to SPELLINGS1 until it is used in that fashion.

Several functions, such as LOAD, use SPLST with a value of 0 so that the function is not added to SPELLINGS2 because there is no indication that the function will (ever) be called by the user.

LISPX invokes ADDSPELL with SPLST set to 1 to handle "apply" format inputs. The input is added to the permanent section of SPELLINGS1 so that it is not forgotten.

LISPX invokes ADDSPELL with SPLST set to 2 to handle "eval" format inputs. The input is added to the permanent section of SPELLINGS2 so that it is not forgotten.

Several functions, such as SAVESET, RPAQ, SETQ, EDITV, and variables given to LISPX, invoke ADDSPELL with SPLST set to 3. The variable names are added to the temporary section of SPELLINGS3.

ADDSPELL1

ADDSPELL1 is the workhorse function used by ADDSPELL. We might define ADDSPELL1 as follows

```
(DEFINEQ
   (addspell1 (word splst index)
 (COND
     ((NULL splst)
         (*
              The spelling list is empty, so create an
              entry for it.
         )
         (SETQ splst (LIST NIL word)))
     ((AND
         (NEQ word (CAR splst))
```

```
            (NEQ word (CADR splst)))
            (*
                The above expressions make a quick check
                of the first two words of the spelling
                list under the assumption that the
                spelling error might have occurred
                recently.
            )
            (PROG (alist1 alist2 alist3 tmplen)
                  (*
                      ALIST1 and ALIST2 are sublists of
                      SPLST that track a search through it
                      so elements may be added or deleted.
                      TMPLEN is the length of the temporary
                      section.
                  )
                  (SETQ alist1 splst)
                  (SETQ alist2 (CDDR splst))
                  (COND
                      ((NULL (CAR splst))
                           (SETQ alist3 splst)
                           (SETQ tmplen 1))
                      ((NULL (cADR splst))
                           (SETQ alist3 splst)
                           (SETQ tmplen 0)))
            loop
                  (COND
                      ((NULL alist2)
                           (*
                               This check occurs because
                               USERWORDS has NIL as its
                               first entry, i.e., there
                               are no permanent user
                               words. If NIL is not
                               noticed, the length will
                               not be determined and
                               nothing will ever be
                               deleted.
                           )
                           (GO exit.addspell1))
                      ((EQUAL word (CAR alist2))
                           (COND
                               ((NULL alist3)
                                    (*
                                        The word is
                                        already in the
```

```
                    permanent
                    section.
               )
               NIL)
          ((NULL index)
               (*
                    The word is in
                    the temporary
                    section. Add it
                    to the permanent
                    section and erase
                    it from the
                    temporary
                    section.
               )
               (RPLACD (CDR alist1)
                       (CDR alist2))
               (RPLACD alist2
                       (CDR alist3))
               (RPLACD alist3
                    alist2))
          (T
               (*
                    The word is in
                    the temporary
                    section. Move it
                    to the front of
                    the temporary
                    section.
               )
               (RPLACD (CDR alist1)
                       (CDR alist2))
               (SETQ alist3
                     (CDR alist3))
               (RPLACD alist2
                       (CDR
                    alist3))
               (RPLACD alist3
                    alist2)))
          (RETURN))
     ((NULL (car alist2))
          (*
               The CAR of alist3 is the
               last member of the
               permanent section.
```

```
                    (SETQ alist3 (CDR alist1))
                    (SETQ tmplen 0)))
            (SETQ alist1 (CDR alist1))
            (SETQ alist2 (CDR alist2))
            (AND tmplen
                 (add1var tmplen))
            (GO loop)
exit.addspell1
            (COND
                ((NULL alist3)
                    (*
                        NIL not found in the
                        spelling list. This occurs
                        if the user is maintaining
                        his own spelling list and
                        not using the temporary and
                        permanent conventions.
                    )
                    (NCONC1 alist1 word))
                ((NULL index)
                    (*
                        Add the word at the end of
                        the permanent section.
                    )
                    (RPLNODE alist3
                             (CAR alist3)
                             (CONS word
                                   (CDR alist3))))
                ((IGREATERP m index)
                    (*
                        Add the word at the
                        beginning of the temporary
                        section. Delete and reuse
                        the last element of the
                        temporary section.
                    )
                    (RPLNODE (CDR alist1)
                             word
                             (CDDR alist3))
                    (RPLNODE (CDR alist3)
                             (CADR alist3)
                             (CDR alist1))
                    (RPLACD alist1 NIL))
                (T
                    (*
```

```
                    Add the word at the
                    beginning of the temporary
                    section.
                )
                (RPLNODE (CDR alist3)
                         (CADR alist3)
                         (CONS word
                               (CDDR
                    alist3)))))))))
        splst))
```

Using ADDSPELL

Many of your programs will create entries on lists as a result of a user entering data to an application. These entries may not be added to a system spelling list because they are not defined by SETQ or DEFINEQ. In these cases, you may wish to invoke spelling correction on a spelling list that is associated with the application. You can do so by creating an atom that will have as its value the spelling list. Then you may add a word to the spelling list by invoking ADDSPELL directly.

```
←(SETQ my.spelling.list
(initialize.spelling.list 'my.spelling.list))
('[spellseparator]' MY.SPELLING.LIST)

←(ADDSPELL 'uganda my.spelling.list 3)
('[spellseparator]' MY.SPELLING.LIST UGANDA)
```

where we have made the name of out spelling list an entry in itself to ensure that it is spelling corrected as well. Consider several more additions:

```
←(ADDSPELL 'kenya my.spelling.list 3)
('[spellseparator]' MY.SPELLING.LIST UGANDA KENYA)

←(ADDSPELL 'egypt my.spelling.list 3)
('[spellseparator]' MY.SPELLING.LIST UGANDA KENYA EGYPT)

←(ADDSPELL 'sudan my.spelling.list 3)
('[spellseparator]' MY.SPELLING.LIST KENYA EGYPT SUDAN)
```

where the oldest name on the spelling list has been forgotten because we specified a length of three words. Now, if we add a new word without the length specification:

```
←(ADDSPELL 'libya my.spelling.list)
('[spellseparator]'   LIBYA MY.SPELLING.LIST KENYA EGYPT
SUDAN)
```

A Definition for INITIALIZE.SPELLING.LIST
We might define INITIALIZE.SPELLING.LIST as follows:

```
(DEFINEQ
     (initialize.spelling.list (x)
          (LIST (CAR USERWORDS)
                x)
     ))
```

where we need to capture the spelling separator.

22.8.2 Finding a Misspelling

MISSPELLED? determines if a word is misspelled. It takes the form

```
Function:       MISSPELLED?

# Arguments:    6

Arguments:      1) a mispelled word, XWORD
                2) a relative agreement, REL
                3) a spelling list, SPLST
                4) a flag, FLAG
                5) an expression for correction, TAIL
                6) a function, FN

Value:          See below.
```

MISSPELLED? operates as follows:

1. If XWORD is NIL or ⟨ESC⟩, MISSPELLED? prints = followed by the value of LASTWORD. It returns LASTWORD as the respelling without asking for approval.
2. MISSPELLED? determines if XWORD is really misspelled. If FN applied to XWORD returns T or XWORD appears on SPLST, then MISSPELLED? returns XWORD.
3. Otherwise, MISSPELLED? attempts to fix the spelling of XWORD via

```
(FIXSPELL xword rel splst flg tail fn tieflg
dontmovetopflg)
```

A Definition for MISPELLED?
MISPELLED? might be defined as:

```
(DEFINEQ
   (mispelled? (xword rel splst flag tail fn)
```

```
(AND DWIMFLG
     (NULL NOSPELLFLG)
     (PROG NIL
           (RETURN
              (COND
                  ((OR (NULL xword)
                       (EQ xword ⟨ESC⟩))
                       (*
                            If you press the
                            ESCAPE key, LISPX
                            interprets this as
                            meaning the last thing
                            typed.
                       )
                       (PRIN1 '= T)
                       (PRINT LASTWORD T T))
                  ((COND
                       ((NULL fn)
                            (*
                                 If no selection
                                 function,
                                 determine if
                                 XWORD exists in
                                 the spelling
                                 list.
                            )
                            (MEMBER xword splst))
                       (T
                            (APPLY* fn xword)))
                       xword)
                  (T
                       (FIXSPELL xword
                                 rel
                                 splst
                                 flag
                                 tail
                                 fn))))))))
```

22.8.3 Fixing the Spelling of a Word

FIXSPELL is the spelling corrector. It takes the form

```
Function:       FIXSPELL
# Arguments:    8
```

```
Arguments:    1) a word to be corrected, XWORD
              2) a relative agreement, REL
              3) a spelling list, SPLST
              4) a flag, FLG
              5) an expression tail, TAIL
              6) a function, FN
              7) a tie flag, TIEFLAG
              8) a movement indicator, DONTMOVETOPFLG

Value:        The respelling of XWORD or NIL.
```

If XWORD is NIL or ⟨ESC⟩, the respelling is the value of LASTWORD. This is returned without asking your approval. This allows you to immediately apply functions such as EDITF and PP that expect a single literal atom as an argument.

If XWORD contains lower case characters, and the corresponding upper case word is correct, the upper case word is returned as its value. The upper case word must be on SPLST or satisfy FN (i.e., it returns T).

REL is a measure of relative agreement used to determine the closeness of two words. If it is NIL, then FIXSPELLREL is used as the default. When you call FIXSPELL directly, you may want to experiment with different values of REL depending on the lengths of words that you will using.

If FLG is NIL, the spelling correction is performed as if you had typed in the word (whether or not you actually did!). In this case, XWORD will not be typed, and your approval will not be sought.

If FLG is T, then XWORD is typed, followed by =, and your approval is requested if APPROVEFLG is T.

If FLG is NO-MESSAGE, the correction is returned with no further processing. It has the same effect as executing CHOOZ. The correction is a pair of words.

If TAIL is non-NIL, and the correction is successful, FIXSPELL makes the correction using /RPLACA (so that is undoable). It also corrects misspellings do to running two words together without an intervening space. In this case, it replaces the CAR of TAIL, where the runon occurred, with two words which contain the intervening space.

If TIEFLAG is NIL, and a "tie" occurs, FIXSPELL returns NIL. A "tie" occurs when two or more words on the spelling list have the same degree of agreement with XWORD. No correction occurs because FIXSPELL cannot wdecide which is the better choice.

You may influence the choice by setting TIEFLAG as follows:

1. If TIEFLAG is PICKONE, the first word found will be used as the spelling correction for XWORD.
2. If TIEFLAG is LIST, FIXSPELL returns a list of the possible corrections for XWORD (i.e., all of the possible candidates). Your program must decide which it likes best and make the appropriate correction.

3. If TIEFLAG is EVERYTHING, a list of all candidates whose degree is above REL will be returned. This includes that tie in degree as well as those which may be better.

If DONTMOVETOPFLG is T, and a spelling correction is made, the corrected word will not be moved to the beginning of the spelling list. In this way, you may prevent the propagation of large numbers of words to the permanent section of the spelling list which the system would consider important, but you surely don't.

Using FIXSPELL

First, let us set up a private spelling list.

```
←(SETQ mywords (LIST (CAR USERWORDS)))
('[spellseparator]')
```

which sets up your own spelling list with the separator in place. (Note: what is needed is a function to create a spelling list).

Now, let us add a few words to the spelling list MYWORDS.

```
←(SETQ #MYWORDS 10)
10
←(ADDSPELL 'YANKEES MYWORDS #MYWORDS)
('[spellseparator]' YANKEES)
←(ADDSPELL 'PIRATES MYWORDS #MYWORDS)
(YANKEES '[spellseparator]' PIRATES)
Now, let us try to correct a word:
←(FIXSPELL 'YANKES 70 MYWORDS)
=YANKEES
YANKEES
←(FIXSPELL 'PRTES 50 MYWORDS)
=PIRATES
PIRATES
```

But, if we increase the relative agreement to 75, we do not obtain a correction:

```
←(FIXSPELL 'PRTES 75 MYWORDS)
NIL
Now, let use the same word with FLAG set to T:
```

```
←(FIXSPELL 'PRTES 50 MYWORDS T)
PRTES = PIRATES? ...yes
PIRATES
```

where we allowed the default time to expire and correction to proceed.

Now, let us add two word forms to the spelling list that have close spellings to each other.

```
←(ADDSPELL 'NEW-YORK-METS MYWORDS #MYWORDS)
(YANKEES PIRATES '[spellseparator]' NEW-YORK-METS)

←(ADDSPELL 'NEW-YORK-JETS MYWORDS #MYWORDS)
(YANKEES PIRATES '[spellseparator]' NEW-YORK-JETS
NEW-YORK-METS)
```

Now, let us attempt to correct the spelling of a word:

```
←(FIXSPELL 'NEW-YORK-ETS 75 MYWORDS)
NIL
```

because more than one word matches the word to be corrected and TIEFLAG is NIL. Now, let us set TIEFLAG to LIST and try again:

```
←(SETQ TIEFLAG 'LIST)
LIST

←(FIXSPELL 'NEW-YORK-ETS 75 MYWORDS NIL NIL NIL TIEFLAG)
(NEW-YORK-JETS NEW-YORK-METS)
```

Now, let us set TIEFLAG to PICKONE and try again:

```
←(SETQ TIEFLAG 'PICKONE)
PICKONE

←(FIXSPELL 'NEW-YORK-ETS 75 MYWORDS NIL NIL NIL TIEFLAG)
=NEW-YORK-METS
NEW-YORK-METS
```

22.8.4 Checking a Function Name Spelling

FNCHECK determines whether or not a word is a function name, and if it is, whether or not its spelling is correct. It takes the form

```
Function:       FNCHECK

#Arguments:     5
```

```
Arguments:      1) a function name, FN
                2) an error flag, NOERRORFLAG
                3) a spelling flag, SPELLFLAG
                4) a property flag, PROPFLAG
                5) a tail expression, TAIL

Value:          The corrected name.
```

FNCHECK is called by many functions to determine if FN is a function (examples are ARGLIST, GETD, SAVEDEF). If FN is a function, then FN is returned as the value of FNCHECK. If FN is not a function, FNCHECK attempts spelling correction if DWIMFLG has the value T. The IRM notes that this allows FNCHECK to operate in small INTERLISP systems where DWIM is not present.

(Note: at the present time, every INTERLISP system has a standard loadup that includes DWIM. Future versions may allow the user to configure his own INTERLISP system. By the time this book is published, Xerox may have announced this feature.)

NOERRORFLAG determines whether or not FNCHECK handles the unsuccessful case. If NOERRORFLG is T, FNCHECK simply returns NIL.

```
←(FNCHECK 'MISPELLD?)
MISPELLD? not a function

←(FNCHECK 'MISPELLD? T)
NIL
```

Otherwise, FNCHECK prints a message and generates a non-breaking error.

SPELLFLAG corresponds to MISSPELLED?'s fourth argument in that it determines whether approval is sought for the spelling correction. MISSPELLED? is called by FNCHECK to correct the spelling of FN.

PROPFLAG is used when FN does not have a function definition, but does have an EXPR property. If PROPFLAG has the value T, FN is considered to be the name of a function and FN will be returned. Otherwise, if PROPFLAG is 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 NOERRORFLAG.

22.8.5 Correcting File Name Spelling

SPELLFILE is used to correct the spelling of a file name. It takes the form

```
Function:       SPELLFILE

# Arguments:    4
```

```
Arguments:      1) a file name, FILE
                2) a print flag, NOPRINTFLAG
                3) a spelling flag, NOSPELLFLAG
                4) a directory list, DIRLST

Value:          The corrected file name; otherwise, NIL.
```

SPELLFILE attempts to correct the spelling of a file name by searching one or more directories for a file name corresponding to FILE.

If FILE has a directory field, SPELLFILE examines the specified directory. Otherwise, it uses the directories specified in DIRLST. If DIRLST is NIL, SPELLFILE uses the value of DIRECTORIES which is initially (NIL LISP). NIL indicates that the user's login directory is to be used as the default.

If NOPRINTFLAG is T, SPELLFILE does not print any correction nor does it ask for your approval of the correction.

If NOSPELLFLAG is T, no spelling correction will be attempted, although searching through DIRECTORIES will be performed.

SPELLFILE is used on ERRORTYPELST to handle FILE NOT FOUND errors. The initial entry is:

```
((23 (SPELLFILE (CADR ERRORMESS) NIL NOFILESPELLFLG)))
```

The value of NOFILESPELLFLG is initially T. But, setting it to NIL will allow SPELLFILE to correct the spelling of the file name whenever an error occurs.

Finding a File

An alternative form, **FINDFILE**, attempts to locate a file in one or more directories. If the file name is misspelled, it calls SPELLFILE to correct the spelling, but with no printing interaction. It takes the form

```
Function:       FINDFILE

# Arguments:    3

Arguments:      1) a file name, FILE
                2) a spelling flag, NOSPELLFLAG
                3) a directory list, DIRLST

Value:          The file name.
```

We might define FINDFILE as follows:

```
(DEFINEQ
     (findfile (file nospellflag dirlst)
          (COND
               ((INFILEP file) file)
```

```
                (T
                     (SPELLFILE file T nospellflag
dirlst)))
     ))
```

22.8.6 A Spelling Correction Example

In this section, we try to show how all of these functions work together to correct a word. We do so by tracing the major functions that we have discussed. We use the small spelling list that was created in our discussion of ADDSPELL above.

```
←(TRACE MISSPELLED?)
(MISSPELLED?)

←(TRACE FIXSPELL)
(FIXSPELL)

←(TRACE CHOOZ)
(CHOOZ)

←(TRACE SKOR)
(SKOR)

←(MISSPELLED? 'SUDNA NIL MY.SPELLING.LIST)
```

where SUDAN is incorrectly spelled as SUDNA. With DWIM enabled in trusting mode, the following trace appears:

```
MISSPELLED?:
XWORD=SUDNA
REL=NIL
SPLST=(SUDAN '[spellseparator]' LIBYA MY.SPELLING.LIST
EGYPT KENYA)
FLG=NIL
TAIL=NIL
FN=NIL
FIXSPELL:
     XWORD=SUDNA
     REL=NIL
     SPLST=(SUDAN '[spellseparator]' LIBYA
MY.SPELLING.LIST EGYPT KENYA)
     FLG=NIL
     TAIL=NIL
     FN=NIL
     TIEFLG=NIL
     DONTMOVETOFLG=NIL
     FROMDWIM=NIL
```

```
        APPROVALFLG=NIL
        CHOOZ:
             XWORD=(83 85 68 78 65)
             REL=70
             SPLST=(SUDAN '[spellseparator]' LIBYA
             MY.SPELLING.LIST EGYPT KENYA)
             TAIL=NIL
             FN=NIL
             TIEFLG=NIL
             NDBLS=0
             FROMDWIM=NIL
             SKOR:
                  XWORD=(83 85 68 78 65)
                  TWORD=(83 85 68 65 78)
                  NCX=5
                  NCT=5
                  FROMDWIM=NIL
             SKOR=0
             CHOOZ=SUDAN
=SUDAN
        FIXSPELL=SUDAN
MISSPELLED?=SUDAN
SUDAN
```

Note that the list of numbers received by CHOOZ is the list of character codes that make up the word to be checked. SKOR receives the character codes for both the word to be checked and the candidate word.

The value of REL has a significant effect on whether or not spelling correction is successful. Consider the example (where I will leave out most of the tracing information):

```
←(MISSPELLED? 'KNYA 90 MY.SPELLIN.LIST)
MISSPELLED?:
XWORD=KNYA
REL=90
...
FIXSPELL:
     XWORD=KNYA
     REL=90
     ...
     CHOOZ:
          XWORD=(75 78 89 65)
          REL=90
          ...
     CHOOZ=NIL
```

```
     FIXSPELL=NIL
MISSPELLED?=NIL
NIL
```

However, if we specify a relative agreement of 50, we obtain the following result (where much of the tracing information is omitted):

```
←(MISSPELLED? 'KNYA 50 MY.SPELLING.LIST)
MISSPELLED?:
XWORD=KNYA
REL=50
...
FIXSPELL:
     XWORD=KNYA
     REL=50
     ...
     CHOOZ:
          XWORD=(75 78 89 65)
          REL=50
          ...
          SKOR:
               XWORD=(75 78 89 65)
               TWORD=(69 71 89 80 84)
               NCX=4
               NCT=5
               ...
          SKOR:
               XWORD=(75 78 89 65)
               TWORD=(83 85 68 65 78)
               NCX=4
               NCX=5
               ...
          ⟨other attempts⟩
          SKOR:
               XWORD=(75 78 89 65)
               TWORD=(75 69 78 89 65)
               NCX=4
               NCT=5
               ...
          SKOR=1
          CHOOZ=KENYA
=KENYA
     FIXSPELL=KENYA
MISSPELLED?=KENYA
KENYA
```

Timing Results for Spelling Correction

Using BREAKDOWN (see Section 29.3), we can obtain some timing estimates for how long it takes to do simple spelling correction. We will use the same spelling list that we used in the previous example:

```
←(BREAKDOWN 'MISSPELLED? 'FIXSPELL 'CHOOZ 'SKOR)
(MISSPELLED? FIXSPELL CHOOZ SKOR)

←(MISSPELLED? 'UGANDI 50 MY.SPELLING.LIST)
=UGANDA
UGANDA

←(BRKDOWNRESULTS)
FUNCTIONS       TIME      #CALLS    PER CALL       %
MISSPELLED?     .012      1         .012           10
FIXSPELL        .018      1         .018           16
CHOOZ           .047      1         .047           41
SKOR            .039      5         .008           34
TOTAL           .116      8         .015
NIL
```

23

Conversational LISP

INTERLISP is unlike any other "traditional" programming language. A number of factors make this so! First, it has a very simple syntax—everything is a list or an atom. If it is a list, then the first element of the list may somehow be interpreted as the name of a function if that list is given to EVAL. What is data at one moment is a function call the next. The structure of a list is enforced through a myriad of parentheses.

Second, INTERLISP has no reserved words. Also, it has function names which have a historical basis that, in many cases, are but a memory. Sometimes, you must think about what a function does because, although a function sounds "right," it is not necessarily right for every instance. This becomes more confusing when you have multiple function names that all sound like they do the same thing but have subtle differences.

INTERLISP programs, even single functions, are difficult to read. Readability in traditional languages is enhanced by syntactic cues such as certain types of statements beginning on a new line or specific punctuation to terminate a statement. INTERLISP has none of these features. When you encounter multiple parentheses, particularly in a nested function definition, it is very hard to keep track of which right parenthesis goes with which left parenthesis.

INTERLISP uses parentheses for almost all syntactic forms for its own benefit rather than the user's (reader's). Thus, there is very little syntactic information carried in the parenthesis notation. Experienced INTERLISP programmers tend to ignore the parentheses and focus on the keywords within the lists that make up INTERLISP functions. After some practice, your eye will become trained to picking out the meaning of a function from its structure (i.e., the form of coding) as opposed to "reading" the function.

However, good programming practice emphasizes that programs should be readable by others as well as oneself. To enhance the readability of INTERLISP programs, Teitleman [teit73] developed Conversational LISP (CLISP) to make INTERLISP programs easier to read and write. CLISP borrows many program-

ming constructs from traditional languages. Usually, users are more familiar with these constructs than with basic INTERLISP formalisms.

CLISP augments INTERLISP syntax rather than replacing it. You may intermix INTERLISP and CLISP syntax freely in a function without having to identify what mode (since INTERLISP is modeless) you are in.

23.1 HOW CLISP OPERATES

CLISP is an integral part of INTERLISP. Thus, you may begin programming in CLISP without any further preparation (other than reading Sections 23.2 through 23.5).

CLISP sits between you and the INTERLISP interpreter. When you type a CLISP statement at the top level, an "error" occurs because the interpreter does not recognize the CLISP syntax. DWIM (see Chapter 22) is called to determine how to process the "error." It transforms the CLISP statement into an equivalent INTERLISP expression and passes it to the interpeter for evaluation. Unlike other errors, you will not be asked to approve the translation and no error message will be generated. Thus, CLISP translation occurs transparently to the user.

CLISP statements are handled by DWIM. When a CLISP statement is encountered, it is unrecognized by the intepreter which calls DWIM to do some sort of correction. DWIM processes the statement, collecting error messages as it goes. Either DWIM successfully translates the statement, whence it is executed directly, or DWIM displays error messages at your terminal.

However, if DWIM encounters a syntactic error in your CLISP statement, it does not complete the translation. Rather, it prints a diagnostic that identifies the error and waits for you to correct it. When DWIM encounters a valid CLISP statement, but one in which an operand is unbound, DWIM asks you whether or not to treat it as CLISP.

Because CLISP statements are intercepted by DWIM and translated to the corresponding INTERLISP forms, you pay a penalty in slower execution due to interpretation of the CLISP expression. You should keep this in mind when timing functions containing CLISP statements. However, this penalty disappears when you compile your functions because all CLISP statements are translated to INTERLISP expressions before compilation.

23.1.1 Translating CLISP Expressions

CLISP expressions are handled by replacing the expression with the corresponding INTERLISP expression. The original CLISP expression will be discarded (actually, it is retained in the hash array CLISPARRAY as a translation). However, the translation will replace the CLISP expression in the function definition unless CLISPRETRANFLG is T.

CLISP statements are discarded because they may easily be recomputed using CLISPIFY. They are also discarded because even though the CLISP statement contained errors, these will be corrected during translation by DWIM.

23.2 CLISP OPERATORS

CLISP recognizes a wide variety of operators that enable you to program more efficiently. These operators may be divided into the list, infix, and prefix classes.

23.2.1 List Operators

CLISP uses **angle brackets**—〈 〉—as a shorthand notation for specifying the list construction. The appearance of a "〈" is interpreted as equivalent to "(". When CLISP encounters a left angle bracket, it begins the construction of a list. Everything up to the matching right angle bracket will be included in the list.

```
← (SETQ old-time-players
        〈 'musial 'wagner 'cobb 'triandos 〉)
(musial wagner cobb triandos)
```

would construct a list equivalent to executing

```
(LIST 'musial 'wagner 'cobb 'triandos)
```

Angle brackets may be nested to indicate sublists of lists as follows:

```
← (SETQ more-players
        〈 'ruth 'maris 〈 'aaron 'robinson 〉〉)
(ruth maris (aaron robinson))
```

is equivalent to

```
(LIST 'ruth 'maris (LIST 'aaron 'robinson))
```

! Operator

The **exclamation point—!**—operator indicates that the following element is to be included in the list as a segment, i.e., when CLISP encounters a !, it performs a CONS to construct a list.

```
←(SETQ double.play.team 〈 'tinker 'evers ! 'chance 〉)
(tinker evers . chance)
```

is equivalent to

```
←(SETQ double.play.team
     (CONS 'tinker (CONS 'evers 'chance)))
```

whereas

```
←(SETQ double.play.team 〈 ! 'tinker ! 'evers 'chance 〉)
(chance)
```

is equivalent to

```
←(SETQ double.play.team
      (APPEND 'tinker 'evers (LIST 'chance)))
```

!! Operator

The **double exclamation point—!!**—operator accomplishes two functions: it indicates that what follows is included in the list as a segment, and that anything in angle brackets to its right is physically attached to it.

```
←(SETQ pitchers ⟨ !! 'koufax 'seivers 'ryan⟩)
(seivers ryan)
```

is equivalent to

```
←(SETQ pitchers (NCONC 'koufax 'seivers 'ryan))
```

and

```
←(SETQ pitchers ⟨ !! 'drysdale ! 'feller ! 'johnson ⟩)
feller
```

is equivalent to

```
←(SETQ pitchers
      (NCONC 'drysdale (APPEND 'feller 'johnson)))
```

Note that the previous form did not translate to

```
(NCONC (APPEND 'drysdale 'feller) 'johnson)
```

which has the same value but the wrong structure. The latter would attach "johnson" to "feller" but not attach either to "drysdale".

These operators—⟨, !, !!, and ⟩—have been described as separate atoms, but they need not be. Moreover, our simple examples do not preclude the use of more complex statements within angle brackets such as complete CLISP statements.

Note that CLISP attempts to match pairs of angle brackets properly. Thus, if a right angle bracket follows a quoted atom, the list structure will not be completed.

```
← ⟨ 'mays 'spahn 'campanella⟩
undefined CAR of form
```

because ⟩ is treated as part of the atom CAMPANELLA. You must separate CLISP operators from other symbols in an input line by at least one space.

23.2.2 Infix Operators

CLISP recognizes several standard arithmetic symbols as shorthand notation for the corresponding INTERLISP functions. The symbols and their corresponding functions are

Symbol	*Function*
+	IPLUS, FPLUS
-	IDIFFERENCE or IMINUS, FDIFFERENCE
*	ITIMES, FTIMES
/	IQUOTIENT, FQUOTIENT
↑	EXPT ↑

Operators have the normal precedence that one expects. As usual, normal precedence may be overridden by proper grouping of arithmetic expressions with parentheses. Consider the following examples:

```
←12*12/6-4+100
120
←(12*12)
144
←210
1024
←0/6
0
←(SETQ x 12)
12
←(SETQ y 100)
100
←(SETQ z 24)
24
←X*Y/Z
50
←(SETQ A (CONS X Y))
(12 . 100)
←((CAR A)*(CDR A)/Z)
50
←"THE LAZY BOY" + "THE BROWN DOG"
NON-NUMERIC ARG
"THE LAZY BOY"
```

The flexibility of the CLISP arithemtic infix operators allows you to perform simple arithmetic computations easily and directly at the top level. You may use literal atoms as the arguments for the CLISP infix operators. You may also use expressions that evaluate to numbers. Note that an extra pair of parentheses is required in the example above because LISPX (see Section 25.2) begins evaluation as soon as its detects a matching, balanced right parenthesis. The arguments are passed to the appropriate arithmetic functions which perform any error detection.

Relational Operators

CLISP also recognizes a shorthand notation for relational operators which is modeled after the Fortran operators. These include

Symbol	*Function*
=	IEQP
GT	IGREATERP
LT	ILESSP
GE	IGREATERP/IEQP
LE	ILESSP/IEQP

Within an S-expression, CLISP will recognize Boolean operators such as AND, OR, MEMBER, and EQUAL. Consider the following examples:

```
←X=Y
NIL
←X LE Y
T
← X * Z GE Y/3
T
```

: Operator

: is an infix operator that specifies the extraction of the substructure of a list.

```
←(SETQ employees '(steve john sue kathy))
(steve john sue kathy)
←employee:4
kathy
```

means to extract the fourth element of the list that is the value of "employee".

:: Operator

:: indicates that the extraction is to be taken with reference to the tail of the list. Thus, assuming a list of the president's last names, we have

```
←presidents::2
(carter reagan)
```

← Operator
← is an infix operator that indicates assignment. This operator is used heavily by the Record Package (see Chapter 27). In its simplest form, x←y is translated as (SETQ x y). By combining ← with : and ::, you may modify any portion of the structure of a list. Thus, x:2←y is interpreted as (replace the second element of X with Y) and becomes (RPLACA (CDR x) y).

The ← operator has a different precedence on the left than on the right. On the left, ← has a higher precedence than other operations except for ' and :. On the right, ← has broader scope, so that a←b+c means a←(b+c).

Note: I have represented the "left arrow" or *assignment* operator as a composition of the two symbols ⟨ and—because most terminals (mine included) do not possess such a symbol on the keyboard.

```
←players←⟨musial ruth cobb wagner⟩
(musial ruth cobb wagner)
```

Warning!!!

Many examples in this text were taken from a Xerox 1100 running INTERLISP-D whose prompt character is a "left arrow." Please be careful not to confuse the two uses of the left arrow.

23.2.3 Prefix Operators

CLISP recognizes several symbols as *prefix operators*. You are already familiar with one operator from the early chapters of this book. This is the ' operator which means to QUOTE its argument. It is now revealed that this is, in fact, a CLISP operator. Consider the CLISP expression

```
x='y        means (EQ x (QUOTE y))
```

Following ', all operators are ignored until the next separator. Thus the expression

```
'x=y        is translated as (QUOTE x=y).
```

" Operator
" is a prefix operator that indicates the logical negation of its argument. It is translated as (NOT argument).

```
"(ASSOC 'disney presidents)
```

is translated as

```
(NOT (ASSOC 'disney presidents)).
```

" may also be used to negate an infix operator. Thus, the expression (x "GE y) is translated as (x LT y).

23.2.4 Operator Precedence

CLISP assigns a *precedence* to each of the operators it recognizes. When many operators and operands are strung together in an expression without the benefit of demarcating parentheses, precedence is required to tell CLISP how to properly translate the expression. The precedences of all the operators are given in the following table (from highest to lowest):

```
Operator Precedence Table
'
:
←    (left precedence)
",-(unary)
↑
*, /
+,-(binary)
←    (right precedence)
=
INTERLISP forms
LT, GT, EQUAL, MEMBER, etc.
AND
OR
IF, THEN, ELSEIF, ELSE
iterative statement operators
```

For a discussion of precedence, consult any good textbook on language theory.

23.2.5 CLISP Declarations

CLISP provides default functions for all of its operators. Thus, the arithmetic operators use either integer or floating point functions as determined by the nature of their arguments. The relational operators use the integer functions as default. The CLISP declarations for each operator are maintained on the property list of the atom representing the operator.

You may declare new interpretations for CLISP operators using **CLISPDEC**, which takes the form

```
Function:       CLISPDEC
# Arguments:    1
Arguments:      1) a declaration list, DECLST
Value:          The value of DECLST.
```

CLISPDEC records the declarations in a global table. The declarations are processed in the order in which they appear in DECLST, so that later declarations will override earlier ones. The initial global declarations are INTEGER and STANDARD. Declaration atoms are taken from the following table.

Declaration	*INTERLISP Functions Used*
INTEGER FIXED	IPLUS, IMINUS, IDIFFERENCE, ITIMES, IQUOTIENT, ILESSP, IGREATERP
FLOATING	FPLUS, FMINUS, FIDFFERENCE, FTIMES, FQUOTIENT, FLESSP, FGREATERP
MIXED	PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, LESSP, GREATERP
FAST	FRPLACA, FRPLACD, FMEMB, FLAST, FASSOC
UNDOABLE	/RPLACA, /RPLACD, /NCONC, /NCONC1, /MAPCONC, /MAPCON
STANDARD	RPLACA, RPLACD, MEMB, LAST, ASSOC, NCONC, NCONC1, MAPCONC, MAPCON

Consider the following examples:

```
←X * Y
1200
←(CLISPDEC '(FLOATING))
(FLOATING)
←X * Y
1200.0
```

Note that CLISP now uses the floating point functions rather than the integer functions to compute the arithmetic result.

Overriding Global Declarations

CLISPDEC places its declarations in a global table. You may override the global declarations for individual functions by placing a local declaration expression

immediately after the argument list of a function. The local declaration expression takes the form

```
(CLISP: . ⟨declarations⟩)
```

Consider the function ADD3 which adds the three numbers given as its arguments. ADD3 is defined using CLISP arithmetic operators as follows:

```
←(DEFINEQ
    (add3 (x y z)
          x + y + z
    ))
(ADD3)

←(ADD3 12 20 34)
66
```

Now, we can override the normal arithmetic functions used to translate the CLISP form by placing a local declaration in the function definition (via ADVISE or the Editor). ADD3 now appears as

```
←(PP add3)
(ADD3
    (LAMBDA (X Y Z)
          (CLISP: . (FLOATING))
          X + Y + Z
))

←(ADD3 12 20 34)
66.0
```

The **CLISP:** statement is converted to a comment of a special form that is represented by CLISP. Whenever a CLISP transformation that is affected by declarations is about to be performed, this comment is searched for a relevant declaration. If one is found, the corresponding function will be used. Otherwise, the global declaration currently in effect will be used. Note that local declarations are effective in the order in which they are given within the CLISP: form.

23.2.6 Operator Definitions

The CLISP operators are defined via properties attached to the atoms representing those operators. Thus, CLISP acts as a table-driven system. The definitions of the operators are given below.

```
←(GETPROPLIST '+)
(CLISTYPE 2 LISPFN IPLUS CLISPCLASS + CLISPCLASSDEF
(ARITH IPLUS FPLUS PLUS))
```

```
←(GETPROPLIST '-)
(CLISPTYPE 7 LISPFN IMINUS UNARYOP T CLISPCLASS-
CLISPCLASSDEF (ARITH IMINUS FMINUS MINUS))

←(GETPROPLIST '*)
(... CLISPTYPE 4 LISPFN ITIMES CLISPCLASS * CLISPCLASSDEF
(ARITH ITIMES FTIMES TIMES))
```

where the ... indicates that there are other properties not associated with CLISP.

```
←(GETPROPLIST '/)
(CLISPTYPE 4 LISPFN IQUOTIENT CLISPCLASS / CLISPCLASSDEF
(ARITH IQUOTIENT FQUOTIENT QUOTIENT))

←(GETPROPLIST ')
(CLISPTYPE 6 LISPFN EXPT)

←(GETPROPLIST '=)
(CLISPTYPE -20 LISPFN EQ CLISPNEG =)
```

Note: You may want to redefine the value of LISPFN to be EQUAL (see my note in Chapter 6).

```
←(GETPROPLIST 'GT)
(CLISPTYPE -20 LISPFN IGREATERP CLISPCLASS GT
CLISPCLASSDEF (ARITH IGREATERP FGTP GREATERP) CLISPNEG
LEQ BROADSCOPE T CLISPISPROP (gt) CLISPISFORM ((X Y) (X
IS GT Y) (X GT Y)))

←(GETPROPLIST 'LT)
(CLISPTYPE -20 LISPFN ILESSP CLISPCLASS LT CLISPCLASSDEF
(ARITH ILESSP LESSP LESSP) CLISPNEG GEQ BROADSCOPE T
CLISPISPROP (lt) CLISPISFORM ((X Y) (X IS LT Y) (X LT
Y)))

←(GETPROPLIST 'GE)
(CLISPTYPE -20 LISPFN IGEQ CLISPCLASS GEQ CLISPCLASSDEF
(ARITH IGEQ GEQ GEQ) BROADSCOPE T)

←(GETPROPLIST 'LE)
(CLISPTYPE -20 LISPFN ILEQ CLISPCLASS ILEQ CLISPCLASSDEF
(ARITH ILEQ LEQ LEQ) BROADSCOPE T)

←(GETPROPLIST 'MEMBER)
(CLISPTYPE -20 CLISPNEG MEMBER BROADSCOPE T CLISPISPROP
(member MEMBERS members) CLISPISFORM ((X Y) (X IS A
MEMBER OF Y) (MEMBER X Y)))
```

and similarly for EQUAL, AND, OR, NOT, and several other INTERLISP functions.

```
←(GETPROPLIST ':)
(CLISPTYPE (14 . 13))

←(GETPROPLIST '←)
(CLISPTYPE (* . -12) LISPFN SETQ)

←(GETPROPLIST '⟨)
(CLISPTYPE BRACKET UNARYOP T CLISPBRACKET (⟨ ⟩ SEPARATOR
! DWIMIFY CLISPANGLEBRACKETS CLISPIFY SHRIEKIFY))
```

The properties and values appearing in these examples will be described in more detail in later sections.

23.3 CONDITIONAL STATEMENTS

CLISP provides conditional structured programming constructs similar to extended Fortran or C. These constructs use the operators IF, THEN, ELSEIF, and ELSE to demarcate the segments of a conditional expression.

The format of a CLISP conditional statement is

```
(IF          ⟨predicate-expression-1⟩
    THEN     ⟨consequent-expression-1⟩
 ELSEIF      ⟨predicate-expression-2⟩
    THEN     ⟨consequent-expression-2⟩
    ELSE     ⟨consequent-expression-3⟩)
```

The general statement form will be translated into a COND expression of the form

```
(COND
    (⟨predicate-expression-1⟩
        ⟨consequent-expression-1⟩)
    (⟨predicate-expression-2⟩
        ⟨consequent-expression-2⟩)
    (T ⟨consequent-expression-3⟩))
```

Conditional statements may be nested as deep as you wish. The predicate and consequent expressions may be any CLISP or INTERLISP expression.

Conditional statements translate into the standard COND expressions. Consider the following function definition:

```
(DEFINEQ
  (to? (node attribute)
       (PROG (offspring)
             (if
                    offspring←(GETPROP node 'offspring)
```

```
                then
                 (RETURN
                   (CDR
                      (ASSOC
                         (CAR (get.inheritance node
                         attribute))
                         offspring)))))
  ))
```

would be translated as

```
(DEFINEQ
   (to? (node attribute)
        (PROG (offspring)
              (COND
                ((SETQ offspring (GETPROP node
                'offspring))
                   (RETURN
                      (CDR
                         (ASSOC
                            (CAR (get.inheritance node
                            attribute)
                            offspring))))))
))
```

In general, the following translations from INTERLISP to CLISP will be observed:

```
IF ⟨x⟩ THEN ⟨y⟩                          (COND
                                          (⟨x⟩ ⟨y⟩))

IF ⟨x⟩ THEN ⟨y⟩ ELSE ⟨z⟩                 (COND
                                          (⟨x⟩ ⟨y⟩)
                                          (T ⟨z⟩))
```

If there is nothing following a THEN or the THEN expression is omitted entirely, the clause will only consist of a predicate expression. You may use lower-case versions of the words IF, THEN, ELSE, and ELSEIF if you prefer.

You may use DWIMIFY (see Section 22.5) to perform the translations from CLISP to INTERLISP and CLISPIFY (see Section 23.8) to translate from INTERLISP to CLISP. Consider the following examples:

```
←(DWIMIFY '(IF X THEN Y))
(COND
     (X Y))
```

```
←(DWIMIFY '(IF X THEN Y ELSEIF X1 THEN Y2 ELSE J3))
(COND
     (X Y)
     (X1 Y1)
     (T J3))
```

23.4 ITERATIVE STATEMENTS

The major power of CLISP is concentrated in allowing the user to write a wide variety of iterative expressions. CLISP supports iterative statements that model all of the popular constructs of traditional programming languages as well as a few new variations.

CLISP iterative statements are usually translated to either a PROG or MAPCAR expression. The specific form used depends on the *iterative statement operators* (called *i.s.oprs*) and the manner in which they are combined. When it translates an iterative expression, CLISP often inserts dummy variables (denoted by preceding $$) as control mechanisms. You will see these dummy variables when you display functions after translation.

For all of the following operators, lower-case versions may be used and will be treated by CLISP as equivalent to the upper-case versions. Lower-case words will improve the readability of your code.

Each iterative statement operator has lower precedence than all INTERLISP forms. Thus, parentheses around the operands may be omitted. CLISP will provide the parentheses internally during the translation process.

23.4.1 I.S.Opr Translation

The exact form of an iterative statement translation depends on the operators used in the expression. If the expression specifies dummy variables (such as: in Y as I from 1 to 10 do PRINT), a PROG will always be used. The general form, using a PROG, is

```
(PROG      ⟨variables⟩
           [initialization]
$$LP
           [eachtime]
           [test]
           [body]
$$ITERATE
           [aftertest]
           [update]
           (GO $$LP)
$$OUT
           [finalization]
           (RETURN $$VAL))
```

where the following conditions hold:

1. **[initialization]** is a sequence of expressions that assign initial values to the loop variables.
2. **[eachtime]** is a sequence of expressions executed each time through the loop that set up the conditions for this iteration.
3. **[test]** is a sequence of expressions that test for loop termination before execution of the [body].
4. **[body]** is a sequence of expressions forming the main function to be performed by the loop.
5. **[aftertest]** is a sequence of expressions that test for termination after loop execution. They correspond to the REPEATWHILE or REPEATUNTIL conditions.
6. **[update]** is a sequence of expressions that increment or otherwise compute new values for the loop variables.
7. **[finalize]** is a sequence of expressions that compute a final value prior to exiting the loop.

$$LP, $$ITERATE, and $$OUT are consistently used in all translations. You may reference them within your CLISP statements. You may also explicitly set or reference the variable $$VAL which contains the value to be returned as the result of the loop.

23.4.2 I.S.Type Operators

An *iterative statement type* operator is one that specifies the action to be taken on each iteration. I.s.type operators have a *body* composed of other iterative statement operators. Each iterative statement must have one and only one i.s.type operator. The i.s.type operators are

```
DO ⟨form⟩          Specifies what is to be done at each
                   iteration. Its value is NIL if there is
                   an explicit or implicit terminating
                   condition. DO without another operator
                   specifies an infinite loop (because
                   there is no terminating condition!). It
                   translates to MAPC or MAPCAR whenever
                   possible.
```

```
(FOR y IN (CDR x)
          DO
               (if
                    y:1 = '**
                then
                    (PRIN1 '**)
```

```
                else
                    (PRIN1 y:1)
                    (SPACES 1))
                (PRIN1 (CDR y))
                (SPACES 1))
```

Transaction processing programs usually have the form of a continuous loop that performs the actions of reading a command, executing the command, and printing the results (just like the INTERLISP intepreter). The structure of such a program might appear as

```
(DO
          (read.command)
          (if
              (execute.command)
           then
              (print.results)
           else
              (GO exit)))
exit .....
```

COLLECT ⟨form⟩ Specifies that the result of evaluating ⟨form⟩ at each iteration is to be collected into a list which is returned as the value of the expression. It is translated to MAPCAR, MAPLIST, or SUBSET as appropriate.

Consider a function which may take either a single atom or a list of atoms as an argument. If a list of atoms is given, the results of the function must be collected into a list to be returned as its value. Here is a fragment from GETFILEPKGTYPE:

```
(FOR x IN type
    COLLECT
      (OR
         (getfilepkgtype x
                         only
                         noerror
                         name)
         (RETURN)))
```

which translates to

```
(PROG ((LST1 TYPE)
      VAL TEM2 TEM1 X)
```

```
$$LP
     (SETQ X
          (CAR (OR (LISTP $$LST1)
                   (GO $$OUT))))
     (SETQ $$TEM1
          (OR
               (GETFILEPACKAGETYPE X
                                   ONLY
                                   NOERROR
                                   NAME)
               (RETURN)))
     (COND
         ($$TEM2
              (FRPLACD $$TEM2
                       (SETQ $$TEM2 (LIST $$TEM1))))
         (T
              (SETQ $$VAL
                   (SETQ $$TEM2 (LIST $$TEM1)))))
 $$ITERATE
     (SETQ $$LST1 (CDR $$LST1))
     (GO $$LP)
 $$OUT
     (RETURN $$VAL))
```

JOIN ⟨form⟩ Specifies that the values resulting from the evaluation of ⟨form⟩ will be NCONCed together. It translates to MAPCON or MAPCONC as appropriate.

GETDEF returns the definition of a name as a type from a file. Consider the following fragment which is used to process the type MACROS:

```
(MACROS
  (CONS 'PUTPROPS
        (CONS name
              (OR (FOR x on (GETPROPLIST name)
                       BY (CDDR x)
                       WHEN
                            (MEMBER (CAR x) MACROPROPS)
                       JOIN
                            (LIST (CAR x)(CADR x)))
                  (GETDEFERR NIL name type options)))))
```

SUM ⟨form⟩ Specifies that the values resulting from the evaluation of ⟨form⟩ at each iteration should be added together.

The IRM suggests that you can define the sum of squares as follows:

```
←(DEFINEQ
   (sum.of.squares (n)
        (FOR i FROM 1 to n SUM (ITIMES i i))
        ))
   (SUM.OF.SQUARES)
```

which, when executed for N equal to 7 would yield the sum of 1 + 4 + 9 + 16 + 25 + 36 + 49 or 140.

```
←(sum.of.squares 7)
    140
```

COUNT ⟨form⟩ — Counts the number of times that the result of evaluating ⟨form⟩ is true.

Suppose I have a vector of numbers whose values represent the closing value of the New York Stock Exchange for 1982. I want to find on how many days the market closed above 1200. I could do so using the following expression:

```
(FOR i from 1 to 365
    COUNT (GREATERP (ELT nyse.close i) 1200.0))
```

ALWAYS ⟨form⟩ — Returns T if every evaluation of ⟨form⟩ is true; otherwise, it returns NIL as soon as NIL is the result of the evaluation. It is equivalent to

```
(EVERY x (FUNCTION ⟨fn⟩))
```

The form (ALWAYS X) is translated as follows:

```
(PROG (($$VAL T))
      $$LP
           (COND
               ((NULL X)
                    (SETQ $$VAL NIL)
                    (GO $$OUT)))
      $$ITERATE
           (GO $$LP)
      $$OUT
           (RETURN $$VAL))
```

NEVER ⟨form⟩ — Returns T if every evaluation of ⟨form⟩ is NIL; otherwise, it returns T as soon as T is the result of the evaluation. It is equivalent to

```
ALWAYS" ⟨form⟩
```

or

```
(NOTANY y (FUNCTION ⟨fn⟩))
```

THEREIS ⟨form⟩ Returns the first value of the iteration variable for which the result of evaluating ⟨form⟩ is non-NIL. It is equivalent to

```
(CAR
    (SOME y (FUNCTION ⟨fn⟩)))
```

23.4.3 I.S.Binding Operators

An *iterative statement binding* operator binds the iteration variable for the iteration. These operators are

FOR ⟨var⟩ Specifies the variable of iteration which is used with an i.s.selection operator. The variable is rebound (if it has the same name as some external variable) for the scope and duration of the iterative statement unless OLD is specified.

FOR ⟨vars⟩ ⟨vars⟩ is a list of variables, the first of which is the iteration variable, while the rest are dummy variables.

Consider the following example from GETDEFFROMFILE which tries to get a definition from a source file.

```
(DEFINEQ
    (getdeffromfile (name type source options)
        (RESETVARS
            (NOT-FOUND-TAG)
            (RETURN
                (FOR file def temp temp2
                 INSIDE
                    (COND
                        ((EQ source 'file)
                            (WHEREIS name type T))
                        (T   source))
                 WHEN
                    (AND ...
```

where TEMP and TEMP2 are bound within various cases in the AND ... clause.

OLD ⟨var⟩ — Specifies that the variable is not to be rebound, e.g., reuse a variable that you have already used.

Consider the example:

```
(FOR OLD (SETQ X ⟨expression⟩)
     BIND (SETQ Y ⟨expression) ...)
```

BIND ⟨var⟩ — Used to specify dummy variables or BIND ⟨vars⟩ to make a variable local to an expression.

23.4.4 I.S.Selection Operators

An *iterative statement selection* operator assigns a value to the iteration variable at each iteration. These operators are used with i.s.type operators described above.

IN ⟨form⟩ — Specifies that the iteration variable is set to successive elements of the value resulting from the evaluation of the form. If no iteration variable has been specified, a dummy will be supplied.

Consider the following example:

```
←(DWIMIFY '(FOR X IN Y JOIN X Y))
    (MAPCONC Y
             (FUNCTION
                   (LAMBDA (X)
                        X
                        Y)))
```

IN OLD ⟨var⟩ — Specifies that the iteration variable is reset to its tail at each iteration.

ON ⟨form⟩ — Specifies that the iteration variable is reset to the corresponding tail at each iteration.

ON OLD ⟨var⟩ — Specifies that the variable is reset to its current value at the end of each iteration (where it is assumed that the variable is set in the body of the expression).

INSIDE ⟨form⟩ Specifies that the iteration proceeds until the first non-list, non-NIL element of the form is encountered.

Consider the following example from INFILEPAIRS in the File Package:

```
(FOR ll IN lst
     DO
          (FOR x INSIDE (CAR ll)
               DO
                    (FOR y INSIDE (CDR ll)
                         DO
                            INFILECOMSVAL (LIST x y))))
```

where the INSIDE expression translates to

```
(PROG ($$VAL X ($$TEM1 (CAR LL)))
$$LP
     (COND
         ((NULL $$TEM1)
              (GO $$OUT))
         ((NLISTP $$TEM1)
              (SETQ X $$TEM1)
              (SETQ $$TEM1 NIL))
         (T
              (SETQ X (CAR $$TEM1))
              (SETQ $$TEM1 (CDR $$TEM1))))
     (INFILECOMSVAL (LIST X Y))
$$ITERATE
     (GO $$LP)
$$OUT
     (RETURN $$VAL))
```

FROM ⟨form⟩ Specifies an initial value for a numerical iteration variable. The iteration variable will automatically be incremented by 1 after each iteration (if no BY clause is specified). If no iteration variable is specified, a dummy variable is supplied and initialized in the translation.

TO ⟨form⟩ Specifies the final value for a numerical iteration variable. The iteration terminates when the value of the iteration variable exceeds the value resulting from

the evaluation of the form (unless some other termination condition is met first).

BY ⟨form⟩ If used with an IN/ON clause, the value resulting from the evaluation of the form determines the tail of the next iteration. The new value of the iteration variable is the CAR of the tail for IN or the tail itself for ON.

BY ⟨form⟩ If used without IN/ON, the value resulting from the evaluation of the form specifies how the iteration variable is set at each iteration. The iteration variable is assumed to be numeric. The new iteration variable value is computed by adding the value of the evaluation of form at each iteration to the current value of the iteration variable.

Consider the combined example for the three operators described above:

```
←(DWIMIFY '(FOR X FROM 1 TO 20 BY 2 DO (PRINT X)))
(PROG (($$TEM1 20)
       $$VAL
       (X 1))
$$LP
     (COND
        ((IGREATERP X $$TEM1)
             (RETURN $$VAL)))
     (PRINT X)
$$ITERATE
     (SETQ X (IPLUS X 2))
     (GO $$LP))
```

AS ⟨variable⟩ Specifies an iterative statement involving more than one iterative variable. The iterative statement terminates when any of the terminating conditions are met. The argument, ⟨variable⟩, specifies the new iterative variable. For the remainder of the statement or until another AS is encountered, all operators will apply to the new iterative variable. Consider the example (after the IRM)

```
(FOR I FROM 1 TO N1
     AS J FROM 1 TO N2 BY 2
        AS K FROM N3 TO 1 BY -1 ...)
```

23.4.5 I.S.Termination Operators

An *iterative statement termination* operator specifies a condition for terminating the iteration. The effect of these operators takes precedence over the termination conditions given by IN/ON ... FROM ... TO ... BY clauses.

WHEN ⟨form⟩ — The i.s.type operator will be executed only if the condition resulting from the evaluation of the form is true.

Consider the following translation:

```
←(DWIMIFY '(FOR X IN Y COLLECT X WHEN (NUMBERP X)))
(SUBSET Y (FUNCTION NUMBERP))
```

UNLESS ⟨form⟩ — The i.s.type operator is executed except when the condition resulting from the evaluation of form is true. It is equivalent to WHEN "⟨form⟩.

Consider the following translation:

```
←(DWIMIFY '(FOR X IN Y COLLECT X UNLESS (NUMBERP X)))
(SUBSET Y
        (FUNCTION
               (LAMBDA (X)
                    (NOT (NUMBERP X)))))
```

WHILE ⟨form⟩ — The i.s.type operator is executed as long as the value resulting from the evaluation of form is true. The form is evaluated before each iteration. A NIL result terminates the iteration. The test is performed before each iteration.

UNTIL ⟨form⟩ — The i.s.type operator is executed as long as the value resulting from the evaluation of the form is NIL. It is equivalent to WHILE "⟨form⟩.

If ⟨form⟩ is a number, then the iteration terminates when the value of the iteration variable is greater than the number.

(UNTIL ⟨x⟩) would translate as

```
(PROG ($$VAL)
$$LP
      (COND
          ((⟨x⟩ (RETURN $$VAL)))
$$ITERATE
      (GO $$LP))
```

REPEATWHILE ⟨form⟩ Like WHILE, except that the test is performed after the iteration, but before the iteration variable is reset so that you may test the value of the iteration variable.

REPEATUNTIL ⟨form⟩ Like UNTIL, except that the test is performed after the evaluation of the body of the iteration, but before the iteration variable is reset so that you may test the value of the iteration variable.

If ⟨form⟩ is a number, then the iteration terminates when the value of the iteration variable is greater than the number.

23.4.6 I.S.Modification Operators

An *iteration statement modification* operator modifies the execution of the iteration statement or the value(s) produced by its execution.

FIRST ⟨form⟩ The form is evaluated once before the first iteration. It is often used to initialize dummy variables (see BIND). Consider the example (after the IRM)

```
(FOR X Y Z IN LST
    FIRST (INITIALIZE Y Z) ...)
```

FINALLY ⟨form⟩ The form is evaluated after the iteration terminates.

Ex: The IRM suggests the following example for counting the number of atoms in a list:

```
(FOR x IN lst
     BIND y←0
          DO
               (IF ATOM x THEN y←y+1)
          FINALLY (RETURN y))
```

EACHTIME ⟨form⟩	The form is evaluated before each iteration. It is often used to assign values to temporary variables or perform other computations which may be used in several places in the body of the iteration.
DECLARE ⟨decl⟩ DECLARE: ⟨decl⟩	Used to insert a declaration after the PROG variable list in the translation or after the LAMBDA expression in a mapping function. Usually used to specify local or special variables for expressions that will utlimately be compiled.
ORIGINAL ⟨i.s.opr⟩	Specifies that the original, built-in interpretation of the iterative statement operator will be used rather than any user-defined interpretations of it.

Each of these operators may have operands that consist of multiple expressions. In the translation, a PROGN will be supplied to handle the multiple operands.

23.4.7 Potential Errors in Iterative Statements

The flexibility of the I.S.OPRs also provides opportunities for generating erroneous expressions. Some pitfalls to watch out for include

1. Operators with null operands, as evidenced by two adjacent operators in an expression:

```
←(DWIMIFY '(FOR X IN Y UNTIL DO (CONS X X)))
error in iterative statement,
missing operand:
... UNTIL ... DO (CONS X X)
```

2. Operands consisting of more than one form (except as explicitly allowed for the i.s.modification operators:

```
←(DWIMIFY '(FOR X IN Y (PRINT X) COLLECT X))
error in iterative statement,
... (Y (PRINT X)) ... COLLECT X
```

3. IN, ON, FROM TO, or BY appear twice or more in the same iterative statement:

```
←(DWIMIFY '(FOR X IN Y AND A IN B DO COLLECT X A))
error in iterative statement,
operator appears twice:
... IN Y AND A IN B DO COLLECT X A
```

4. Both IN and ON used on the same iterative variable:

```
←(DWIMIFY '(FOR X IN Y AND A ON Y DO COLLECT X A)))
error in iterative statement,
can't use both of these operators together:
... IN Y AND A ON Y DO COLLECT X A
```

5. FROM or TO used with IN or ON on the same iterative variable:

```
←(DWIMIFY '(FOR X IN Y DO FROM 1 TO Y (ADD1 X)))
error in iterative statement,
can't use both these operators together:
... IN Y DO FROM 1 TO Y (ADD1 X))
```

6. Missing DO, COLLECT, or JOIN:

```
←(DWIMIFY '(FOR X IN Y (ADD1 X)))
(MAPC Y (FUNCTION ADD1))
```

where CLISP has inserted a DO after Y. On the other hand, consider

```
←(DWIMIFY '(FOR X IN Y CONS X))
No DO, COLLECT, or JOIN in:
(FOR X IN (Y CONS X))
```

7. No terminating condition is detected:

```
←(DWIMIFY '(FOR X FROM 1 BY 3 DO (PRINT X)))
Possible non-terminating iterative statement:
(FOR X FROM 1 BY 3 DO (PRINT X))
```

```
(PROG ($$VAL (X 1))
$$LP
     (PRINT X)
$$ITERATE
     (SETQ X (IPLUS X 3))
     (GO $$LP))
```

where there is no way to terminate the PROG loop.

The iterative statement is still translated because you may terminate the statement from within a function invoked in the statement by a RETFROM, by inducing an error (such as in the Editor), or executing ⟨CTRL-E⟩.

When an erroneous condition is detected, a diagnostic message will be printed unless CLISP.I.S.GAG is T. It is initially NIL. When an error occurs, the iterative statement is not changed (e.g., translated).

23.4.8 Defining New Iterative Statement Operators

You may define a new I.S.OPR or redefine an existing one using **I.S.OPR**. It takes the form

```
Function:       I.S.OPR
# Arguments:    4
Arguments:      1) an i.s.opr name, NAME
                2) an expression, EXPRESSION
                3) an optional list of i.s.oprs, OTHERS
                4) an evaluation flag, EVALFLAG
Value:          The name of the new i.s.opr.
```

If EXPRESSION is a list, then NAME is the name of the new i.s.opr and EXPRESSION is its body, e.g., its translation. For example, we might define GATHER similarly to COLLECT as follows:

```
←(I.S.OPR 'GATHER '(SETQ $$VAL (NCONC1 $$VAL BODY)))
GATHER
←(DWIMIFY '(FOR X IN Y GATHER X))
(PROG (($$LST1 Y)
       $$VAL X))
 $$LP
     (SETQ X
          (CAR (OR (LISTP $$LST1)
                   (GO $$OUT))))
     (SETQ $$VAL (NCONC1 $$VAL X))
```

```
$$ITERATE
     (SETQ $$LST1 (CDR $$LST1))
     (GO $$LP)
$$OUT
     (RETURN $$VAL))
```

The IRM notes that you may define the following versions of COLLECT that use different functions:

```
(I.S.OPR 'RCOLLECT
         '($$VAL←(CONS BODY $$VAL))
         '(FINALLY (RETURN (DREVERSE $$VAL))))
```

RCOLLECT uses CONS instead of NCONC1 and reverses the values that it collects. Here, OTHERS is an list of additional i.s.operators and operands that will be inserted in the iterative statement where NAME appears.

EXPRESSION may be NIL, whence the new i.s.opr is defined by OTHERS.

```
(I.S.OPR 'TCOLLECT
         '(TCONC $$VAL BODY)
         '(FIRST $$VAL←(CONS) FINALLY (RETURN (CAR
         $$VAL))))
```

In EXPRESSION and OTHERS, you may use the atom $$VAL to refer to the value returned by the iterative statement. You may use I.V. to refer to the current iterative variable. And you may use BODY to refer to the operand of NAME.

If EVALFLAG is T, EXPRESSION and OTHERS are evaluated at translation time and the value is used in the translation. This allows you to vary the translation for the same i.s.opr depending on conditions within your program.

You may save new I.S.OPRs that you have defined using the File Package command I.S.OPRS (see Section 17.2.15).

23.5 ENGLISH PHRASES

One objective of CLISP is to make INTERLISP code more readable. To this end, it accepts a limited number of English-like constructions of the form "⟨subject⟩ is ⟨object⟩".

```
NROWS is a NUMBER
(MAKEFILENAME x) is not a STRING
```

Both the subject and the object may be distributed:

```
XWORD AND TWORD ARE ATOMIC
```

means

```
XWORD IS ATOMIC and TWORD IS ATOMIC
```

CLISP converts INTERLISP expressions into "english" when CLISPIFYENGLSHFLG is T.

23.5.1 Basic English Forms

CLISP knows about a set of English words in both their singular and plural forms. This set includes

```
ARRAY      ATOM        ATOMIC      FLOATING POINT NUMBER
LIST       LITATOM     NIL         LITERAL ATOM
NULL       NUMBER      STRING      SMALL INTEGER
EQ TO      EQUAL TO    GEQ         SMALL NUMBER
GT         LT          MEMB OF     GREATER THAN
TAIL OF    MEMBER OF   LESS THAN
```

Any relationship may be negated by using NOT or N'T:

```
(CADR NAME) AND (CDDR Y) AREN'T LISTS.
```

23.5.2 Defining New Words

You may expand the lexicon used by CLISP using the function **NEWISWORD**, which takes the form

```
Function:       NEWISWORD

# Arguments:    4

Arguments:      1) a singular form, SING
                2) a plural form, PLURAL
                3) a translation form, FORM
                4) the form parameters, VARS

Value:          The name of the new word that will be
                recognized by CLISP.
```

For example, the IRM notes that you may define the phrase SMALL INTEGER using the following expression:

```
←(NEWISWORD
     '(X IS A SMALL INTEGER)
     '(ARE SMALL INTEGERS)
```

```
      '(SMALLP X)
      '(X))
SMALL

←(DWIMIFY '(5 IS A SMALL INTEGER))
(SMALLP 5)

←(5 IS A SMALL INTEGER)
5
```

Note that the plural form does not include the subject.

The ability to define new words for CLISP translations allows you to extend your interactive environment to accommodate new sentences. In fact, you may define a subset of English using appropriate NEWISWORD and I.S.OPR definitions that will make your programs much more accessible.

23.6 CLISPIFYING

CLISPIFY is a function that converts an INTERLISP expression to CLISP notation. It takes the form

```
Function:       CLISPIFY

# Arguments:    2

Arguments:      1) an expression, EXPRESSION
                2) an optional list, LST

Value:          The translation of EXPRESSION.
```

If EXPRESSION is an atom and LST is NIL, EXPRESSION is assumed to be the name of a function. CLISPIFY operates on its definition (via GETD) or its EXPR property value. After CLISPIFYing, EXPRESSION is redefined via /PUTD with its new CLISP definition.

If EXPRESSION is atomic, but not the name of a function, CLISPIFY attempts to correct its spelling. If this is successful, it proceeds as above. Otherwise, an error is generated.

```
←(CLISPIFY '(PROG (X) (COND ((NULL X) (SETQ X (CAR Y)))
(T (PRINT X)))))
(PROG (X) (if X=NIL then X←Y:1 else (PRINT X)))
```

If X is an S-expression and LST is not NIL, X will be translated to CLISP notation. LST is an edit push-down list that provides a context for DWIMIFYing the expression.

```
←(CLISPIFY '(itimes x y))
(x*y)
```

23.6.1 CLISPIFY Variables

CLISPIFY operation is affected by the values of several global variables.

CL:FLG — Affects the translation of CAR, CDR, CADR, ..., CDDDDR to the corresponding infix notation using the : operator. If CL:FLG is T, simple expressions where the argument is atomic or a simple list will be converted to : notation. If CL:FLG is ALL, every possible conversion to : notation will be made. For example,

```
←(SETQ nfl.coaches '(gibbs landry grant stram))
(gibbs landry grant stram)
←(SETQ CL:FLG T)
T
←(CLISPIFY '(SETQ retired.coaches (CADR nfl.coaches)))
(retired.coaches←nfl.coaches::2)
```

CLREMPARSFLG — Affects the removal of parentheses from simple expressions in order to improve their readability. It initially has the value NIL. For example,

```
←(SETQ CLREMPARSFLG T)
T
←(CLISPIFY '(COND ((ATOM X) (LIST X)) ((LISTP X) X) (T
(CONS X NIL))))
(if ATOM X then X elseif LISTP X then X:1 else (CONS X
NIL))
```

CLISPIFYPACKFLG — Affects the form of the translation for atomic operands. If its value is T, a translation involving infix operators can be packed into a single atom. For example,

```
←(SETQ CLISPIFYPACKFLG NIL)
NIL
←(CLISPIFY '(ITIMES X Y))
(X * Y)
```

```
←(SETQ CLISPIFYPACKFLG T)
T
←(CLISPIFY '(ITIMES X Y))
(X*Y)
```

CLISPIFYENGLSHFLG Affects the production of English phrases. When it is T, CLISPIFY translates INTERLISP expressions into their corresponding English phrases using either the SYSHASHARRAY or CLISPARRAY. For example,

```
←(SETQ CLISPIFYENGLSHFLG T)
T
←(CLISPIFY '(COND ((ATOM X) X) ((LISTP X) (CAR X)) (T
(PRIN1 X))))
(if (ATOM X) then X elseif (LISTP X) then X:1 else (PRIN1
X))
```

FUNNYATOMLST Allows CLISP to avoid conflicts between its translations and atoms defined by the user. For example, if you have defined atoms of the form X, Y, and X*Y (a legal atom!), then

```
←(CLISPIFY '(ITIMES X Y))
X*Y
```

which conflicts with a variable name. You may prevent this conflict by adding the atom X*Y to FUNNYATOMLST. CLISPIFY will produce

```
←(SETQ funnyatomlst (LIST 'x*y))
(x*y)
←(CLISPIFY '(ITIMES X Y))
(X * Y)
```

Note: You should avoid using CLISP operators as components of variable names. In general, there is no reason to use such meaningless variable names when you have the ability to make variable names mnemonically interesting.

CLISPIFYPRETTYFLG Affects the prettyprinting of all expressions. If its value is T, PRETTYPRINT invokes CLISPIFY on all expressions before printing them. MAKEFILE (see Section 17.3.1) automatically sets CLISPIFYPRETTYFLG to T before writing expressions to a file.

It takes the following values:

ALL	Apply to all functions
T.EXPRS	Apply to functions that are currently defined as EXPRs
CHANGES	Apply to functions that have been marked as changed
⟨list⟩	Apply only to functions that are mentioned in the list

23.7 CLISP CONVENTIONS

CLISP acts like a table-driven translator. Each operator has properties which describe how its translation is to be affected. You may add new operators or redefine existing operators by modifying the values of properties associated with those operators. CLISP uses the following properties:

CLISPTYPE This property is the precedence of the operator. It defines how the operator will be translated relative to other operators. The actual value is not important, only its value relative to other operators. The value may be a number, a dotted pair, or an expression to be evaluated.

An operator may have different left and right precedence by setting the value of CLISPTYPE to a dotted pair of two numbers. For example, the precedence of ← is the dotted pair (8. -12). The CAR of the dotted pair is the left precedence while the CDR is the right precedence. The IRM notes that A*B←C+D is parsed as A*(B←(C+D)) because the left precedence of ← is 8 while that of * is 4, but its right precedence is -12 which is lower than that of + which is 2.

The precedences of the CLISP operators are given in Section 23.2.4.

Removing the precedence of a CLISP operator disables the corresponding transformation (see CLDISABLE).

UNARYOP — This property with value T declares the operator to be a unary operator. Its operand must always be to its right in the expression.

BROADSCOPE — This property with value T declares that the operator has lower precedence than INTERLISP forms. For example, (FN1 X AND Y) would be translated as ((FN1 X) AND Y) if the BROADSCOPE property for AND is T, but (FN1 (X AND Y)) otherwise.

LISPFN — This property has as its value the INTERLISP function to which the operator will be translated. The translations for the CLISP operators are given in Section 23.2.6.

If LISPFN is NIL, the operator is also the function itself. For example, AND is both a CLISP operator and the corresponding function.

SETFN — If an operator has a SETFN property, then an expression of the form

(⟨op⟩ ...) ← ⟨expression⟩

will be translated to an expression that sets the corresponding values.

For example, you may make # be a CLISP operator equivalent to the function ELT (see Section 11.2.6) by putting the CLISPTYPE property on its property list. If you put a SETFN property on the property list of #, expressions of the form X#N ← Y will be translated to (SETA X N Y).

CLISPINFIX — This property is used when translating INTERLISP forms to infix notation. This property resides on the property list of the INTERLISP function. For example,

```
←(GETPROP 'EXPT 'CLISPINFIX)
↑

←(GETPROP 'TIMES 'CLISPINFIX)
(CLISPINFIX *)
```

CLISPWORD This property is associated with CLISP operators that may be the CAR of an expression. Its form is

(⟨keyword⟩ . ⟨name⟩)

where ⟨name⟩ is the lower-case version of the operator and ⟨keyword⟩ is its type.

```
←(GETPROP 'FETCH 'CLISPWORD)
(RECORDTRAN . fetch)
```

⟨keyword⟩ may also be a function. When the atom appears as the CAR of an expression, the function is either applied to the expression to change the expression or calls CLISPTRAN to store a translation.

CLISPBRACKET This property defines new bracket operators. It must appear on the property list of both the left and right brackets.

The property must also appear on the property list of any functions that the expressions involving the brackets will translate to. The value of the property in this case is a list whose CAR is the left bracket symbol, whose CADR is the right bracket symbol, and whose CDDR is a list in property list format which may include the properties

DWIMIFY	Value is a function to be called to DWIMIFY the construct. It is given the DWIMIFYed expression bounded by the brackets.
CLISPIFY	Value is the function to be called when CLISPIFYing the construct.

SEPARATOR	Value is a character to be split from any atoms, but otherwise unprocessed. For example, for ⟨ and ⟩ the separator is !.

To define [and] as brackets which translate to ELT in the case of a single expression and ELTM for more than one:

```
←(PUTPROP '[ 'CLISPTYPE 'BRACKET)
BRACKET

←(PUTPROP '] 'CLISPTYPE 'BRACKET)
BRACKET

←(PUTPROP   '[ 'CLISPBRACKET '([ ] (SEPARATOR NIL
DWIMIFY FOO))
'([ ] (SEPARATOR NIL DWIMIFY FOO))

←(PUTPROP '] 'CLISPBRACKET '([ ] (SEPARATOR   NIL
DWIMIFY FOO))
'([ ] (SEPARATOR NIL DWIMIFY FOO))

←(DEFINEQ
  (FOO (EXPRESSION LST)
       (IF (MEMBER ', LST)
       THEN
          ⟨'ELTM EXPRESSION
                 (FOR X IN LST
                          COLLECT X
                          WHEN X=',)⟩
       ELSE ⟨'ELT EXPRESSION ! LST⟩)))
(FOO)

←(NCONC CLISPCHARS '([ ] ,))
(↑ * / + - = ← : ' +- = ⟨ ⟩ @ ! [ ])

←(SETQ CLISPCHARRAY (MAKEBITTABLE CLIPSCHARS))
{ARRAYP}]#542224
```

Then, X:1[N-1] will dwimify to (ELT (CAR X) (SUB1 N)) and Z[N,M] will dwimify to (ELTM Z N M).

To enable CLISPIFYing, you must define

```
←(PUTPROP 'ELT 'CLISPBRACKET '[)
[
```

```
←(PUTPROP 'ELTM 'CLISPBRACKET '[)
[
Then, (ELTM (CADR A) (IQUOTIENT (SUB1 N) 2)) I) will
clispify to A:2[(N-1)/2,I]
```

23.8 CLISP FUNCTIONS

CLISP provides a number of functions that allow you to produce the effect of a CLISP translation from within your program or modify the state of your environment.

23.8.1 CLISPIFYing Functions

CLISPIFYFNS is an NLAMBDA, nospread function that takes a list of one or more functions as its argument. It invokes CLISPIFY on each function in the list. It takes the form

Function:	CLISPIFYFNS
# Arguments:	1
Argument:	1) a function specification, FNS
Value:	A list of functions that have been CLISPIFYed.

CLISPIFYFNS handles the following cases:

1. If FNS is an atom and is the name of a function, then that function is CLISPIFYed.
2. If FNS is an atom and is not the name of a function, CLISPIFYFNS determines if it is the name of a file. If it is, then the list of function given by the FNS command in the File Package commands is used as the list of functions to CLISPIFY.
3. If FNS is neither the name of a function nor the name of a file, it generates an error.
4. If FNS is a list, then CLISPIFY is applied to each element of the list.

If any element of the list does not have a function definition, CLISPIFYFNS simply moves to the next element. The application of CLISPIFY is errorset protected so that a non-function name in the middle of the list will not cause the operation to be terminated.

23.8.2 Disabling CLISP Operators

You may disable a CLISP operator by executing the function **CLDISABLE**, which takes the form

```
Function:       CLDISABLE
# Arguments:    1
Argument:       1) an operator, OPERATOR
Value:          The operator.
```

Disabling a CLISP operator causes the operator to be treated as just another character. Consider the following example:

```
←5*10
50
←(CLDISABLE '*)
*
←5*10
UNBOUND ATOM
5*10
```

23.8.3 Storing CLISP Translations

For frequently used expressions, you may *cache* the CLISP translation of an expression for future use by executing **CLISPTRAN**, which takes the form

```
Function:       CLISPTRAN
# Arguments:    2
Arguments:      1) an expression, EXPRESSION
                2) a translation, TRANSLATION
Value:          The value of the translation.
```

CLISPTRAN stores TRANSLATION as the translation of EXPRESSION in the hash array CLISPARRAY. CLISPTRAN is usually called by CLISPIFY to preserve translations. Consider the following example:

```
←(CLISPTRAN 'X*Y '(ITIMES X Y))
(ITIMES X Y)
```

23.9 CLISP VARIABLES

CLISP uses many global variables to drive the translation process. These variables are available for inspection and modification by your programs so that you may determine how your program will be translated.

CLISPFLG — Determines when CLISP translations are performed. It takes one of three values:

1. If CLISPFLG is NIL, no CLISP infix or prefix translations will take place. However, IF/THEN/ELSE and iterative statements will not be affected.
2. If CLISPFLG is TYPE-IN, CLISP translations are performed only on expressions typed in by you (including DEFINEQ expressions), but not on existing functions that you have already defined.
3. If CLISPFLG is T, CLISP translations are performed on all expressions regardless of their origin.

The initial value of CLISPFLG is T.

If you CLISPIFY any expression, it will leave CLISPFLG set to T (a side effect).

CLISPARRAY — A hash array (see Section 11.3) that is used to stored all CLISP translations. Both FAULTEVAL and FAULTAPPLY check CLISPARRAY for the translation of erroneous forms before calling DWIM.

CLISPHELPFLG — If NIL, DWIMIFY (see Section 22.5) will not ask your approval for any CLISP translations. Instead, it assumes a default of "No". IF CLISP is enabled, this flag should be set to T.

CLISPIFTRANFLG — If NIL, CLISP replaces IF/THEN/ELSE statements by the corresponding COND expressions. If T, the translations are stored in a system table. It is initially NIL.

CLISPRETRANFLG If T, DWIMIFY will (re)translate all expressions having translations stored in the hash array or in CLISP% format. It is initially NIL.

CLISPIFYUSERFN If T, its value is applied to each expression which is not recognized by CLISPIFY. That is, if CLISP does not have a translation for an expression, the function given by CLISPIFYUSERFN is called to deal with the expression. If a non-NIL value is returned, it it assumed to be the CLISPIFYed expression.

23.10 THE CHANGETRAN PACKAGE

The CHANGETRAN Package is a mechanism for expressing, in a shorthand notation, some of the more commonly used structure modification operations. CHANGETRAN defines a set of CLISP words that encode the kind of modification that will be performed when the word is encountered and translated by INTERLISP.

23.10.1 CHANGETRAN Words

The CHANGETRAN Package defines a set of basic words that are recognized by CLISP:

(ADD ⟨datum⟩ ⟨item1⟩ ... ⟨itemN⟩)
The values of the specified items are added to ⟨datum⟩ and the result is stored back into ⟨datum⟩. CLISP uses IPLUS, FPLUS, or PLUS depending on the CLISP declarations currently in effect. Consider the following examples:

```
←(SETQ X 100)
100
←(ADD X 25)
125
```

(PUSH ⟨datum⟩ ⟨item1⟩ ... ⟨itemN⟩)
The values of the specified items are CONSed onto the front of the current value of ⟨datum⟩. The result is stored as the value of ⟨datum⟩. Consider the following examples:

```
←(SETQ STACK NIL)
NIL
```

```
←(PUSH STACK 'A)
(A)
←(PUSH STACK 'B)
(B A)
```

(PUSHNEW ⟨datum⟩ ⟨item⟩)
The specified item is CONSed onto the front of the current value of ⟨datum⟩ if and only if it is not already a member of ⟨datum⟩'s value. Consider the following examples:

```
←(PUSHNEW STACK 'A)
(B A)
←(PUSHNEW STACK 'C)
(C B A)
```

(PUSHLIST ⟨datum⟩ ⟨item1⟩ ... ⟨itemN⟩)
The specified items are APPENDed rather than CONSed to the front of the current value of ⟨datum⟩.

(POP ⟨datum⟩)
The first element of ⟨datum⟩ is returned. The CDR of the original value of ⟨datum⟩ is stored as the new value of ⟨datum⟩. Consider the following examples:

```
←(POP STACK)
C
←STACK
(B A)
...
←(POP STACK)
A
←(POP STACK)
NIL
```

(SWAP ⟨datum1⟩ ⟨datum2⟩)
The values of ⟨datum1⟩ and ⟨datum2⟩ are assigned to each other. Consider the following example:

```
←(SETQ X 100)
100
←(SETQ Y 200)
200
```

```
←(SWAP X Y)
200
←X
200
←Y
100
```

⟨CHANGE ⟨datum⟩ ⟨form⟩⟩
The value of ⟨form⟩ is an arbitrary expression that enables you to specify what the new value of ⟨datum⟩ should be. The new value of ⟨datum⟩ is the value of ⟨form⟩* where ⟨form⟩* is constructed from ⟨form⟩ by substituting the value of ⟨datum⟩ for every occurrence of the literal atom DATUM in ⟨form⟩.

23.10.2 Defining New CLISP Words

You may define new CLISP words for use by the CHANGETRAN Package by placing the property CLISPWORD on the atom representing the word. The value of the property has the form

```
(CHANGETRAN . ⟨word⟩)
```

You must put the property on both the upper- and lower-case versions of the word. On the lower-case version of the word, you must place the property CHANGEWORD with a function as its value. The function will be applied to a single argument, the expression which is the argument to the word. It must return an expression that CHANGETRAN can translate into an appropriate expression.

The expression returned from the function should include the literal atom DATUM wherever you want to access the current value of the datum that appears as the first argument to the word. The form (DATUM← ⟨expression⟩) must appear only once in the expression. It specifies that an appropriate expression for storing into the datum should occur at that point.

The IRM suggest the following definition for the CHANGEWORD SUB:

```
←(PUTPROP 'sub
        'CHANGEWORD
        '(LAMBDA (form)
           (LIST 'DATUM←
                 (LIST 'DIFFERENCE
                       'DATUM
                       (CONS 'IPLUS (CDDR FORM))))))
(LAMBDA (FORM) (LIST (QUOTE DATUM←) (LIST (QUOTE
DIFFERENCE) (QUOTE DATUM) (CONS (QUOTE IPLUS) (CDDR
FORM)))))
```

```
←(PUTPROP 'SUB 'CLISPWORD '(CHANGETRAN . SUB))
(CHANGETRAN . SUB)
```

Note that you do not have to define the lower-case version if you do not want to! Personally, since INTERLISP generally believes in upper-case syntax, I find it difficult to use lower-case words.

```
←(SETQ X 100)
100
←(SUB X 25)
75
```

The CHANGETRAN facility allows you to define new languages for interacting with INTERLISP and with your programs. By defining "words" and their translations, you may create different interfaces for different users.

24

INTERLISP User's Packages

INTERLISP provides many utility packages which cannot be included in the standard loadup (e.g., the initial SYSOUT file) due to virtual address space limitations. This is true whether you are running under INTERLISP-10, INTERLISP/370, or INTERLISP-D. This chapter will survey some of these packages and describe some of the functions included therein. Because of the large number of packages provided by INTERLISP-10 (and the even larger number provided by INTERLISP-D), I can only briefly discuss some of the more salient features of many of these packages. In general, additional documentation may be found in files on the respective file systems supporting your INTERLISP environment.

I will discuss only those packages which work on any of the versions of INTERLISP. Packages specific to a version (such as INTERLISP-10) are described in the IRM which you should consult for more information.

24.1 FILE SYSTEM EXTENSIONS

Many of the LISP User's packages provide additional functions that extend the capabilities of the file system or the File Package. These packages are discussed in the following sections.

24.1.1 Indexing a File

SINGLEFILEINDEX is a package for alphabetically indexing the functions found in a single file. The function index is placed at the beginning of the source file. An index consists of a number of columns whose entries are a file name and a byte offset into the file. The number of columns is determined by the length of the longest function name and the number of functions within the file. Moreover, each function is preceded by its function right-justified on a page.

SINGLEFILEINDEX, when loaded, modifies the execution of LISTFILES1, a worker function called by LISTFILES (see Section 17.3.4).

SINGLEFILEINDEX includes the function SINGLEFILEINDEX which performs the indexing and printing to the output file. It takes the form

```
Function:       SINGLEFILEINDEX
# Arguments:    3
Arguments:      1) a source file, SRCFILE
                2) a destination file, DSTFILE
                3) a new page flag, NEWPAGEFLG
Value:          The source file name.
```

SRCFILE must be a source code file. DSTFILE must be the name of a file or NIL. If it has the same value as SRCFILE, then a new version of SRCFILE will be created (the normal case). If DSTFILE is NIL, the output will be directed to the global variable PRINTER which is usually initialized to the name of the line printer for your INTERLISP system.

NEWPAGEFLG determines whether each function will be printed beginning on a new page. If NEWPAGEFLG is T, this will be done. It usually has the value NIL.

SINGLEFILEINDEX uses the following variables:

FILELINELENGTH	To determine how to right-justify function indices and to columnate within the file.
PRINTER	A global variable containing the name of the psuedo-file corresponding to the system printer.
LINESPERPAGE	The number of lines per page that are normally allowed by the printer, so it may determine when to insert page breaks for nice formatting. The initial value is 58.

The compiled code for SINGLEFILEINDEX resides in the file SINGLEFILEINDEX.COM.

24.1.2 Indexing Multiple Files

MULTIFILEINDEX is a package for indexing a set of files which comprise all or part of a system. When you have a large number of files comprising a system, the listings representing the system aren't much use because you have no way of

finding your way around them. The problem is particularly acute if you are not intimately familiar with the structure and implementation details of the system. MULTIFILEINDEX can assist you by creating an alphabetized table of contents of the files with all functions indexed and other data structures (such as records, blocks, etc.) appropriately noted.

MULTIFILEINDEX requires that the COMS of each file to be processed be loaded into memory. It uses the function GETDEF to obtain the definition of variables from the source file. In many files, there may be several levels of indirection. For example, if the list of functions contained within the file is supplied as the value of a variable, say TESTFNS, with the COMS entry appearing as (FNS * TESTFNS), a GETDEF to the COMS entry does not obtain all of the required information. MULTIFILEINDEX will perform a GETDEF to obtain the value of TESTFNS as well.

MULTIFILEINDEX contains a single function that is invoked by the user, MULTIFILEINDEX. It takes the form

```
Function:       MULTIFILEINDEX
# Arguments:    3
Arguments:      1) a list of source files, SRCFILES
                2) a destination file, DSTFILE
                3) a new page flag, NEWPAGEFLG
Value:          A list of files processed.
```

SRCFILES may either be a list of files (such as that created by FILES?) or an atom. If it has the value NIL, MULTIFILEINDEX returns immediately. If it has the value T, the value of FILELST is used. If it is an atom, but not NIL or T, then FILEFNSLST is applied to its value under the assumption that it is the name of a file.

DSTFILE is the destination file where the output from MULTIFILEINDEX will be written. For large numbers of files, the output can be voluminous. If DSTFILE has the value NIL, the value of the global variable PRINTER is used (it stores the name of the system printing device).

NEWPAGEFLG indicates whether each function in the listing should be printed on a page by itself. Giving it the value of T enables this action.

MULTIFILEINDEX produces an output file that contains

1. A listing of all the functions in a file with each function being preceded by its index number within the file right-justified on the line.
2. An alphabetized table of contents indicating the name of each INTERLISP object or entity, the file that it belongs to, and its type. If the entity is a function, the information also includes a unique index in the listing, its type, and its argument list.

In addition, each file that is treated by MULTIFILEINDEX will have its name removed from NOTLISTEDFILES.

Header information will be placed at the top of each page. Each page is numbered. Page width is controlled by FILELINELENGTH while page length is controlled by LINESPERPAGE. LINESPERPAGE initially has the value 58 as described in Section 24.1.1.

MULTIFILEINDEX formats the columns of its report according to the values of four global variables:

MULTIFILEINDEXCOLS	Indicates how the other three columns are to be interpreted. If it has the value FLOATCOLS, which is the default, MULTIFILEINDEX attempts to arrange the columns on the page such that the amount of space for the type information is maximized. This space must be at least 45% of FILELINELENGTH. If it has the value T or FOXCOLS, the values of the other variables indicate absolute column positions on the page. If it has the value NIL or FIXFLOATCOLS, the columns will be floated on the page, but will not be any smaller than the columns defined by the values of the other variables (e.g., you may specify minimum size columns).
MULTIFILEINDEXNAMECOL	Specifies the position of the name column in the report. Initially, its value is 0.
MULTIFILEINDEXFILECOL	Specifies the position of the file name column. Initially, its value is 26.

MULTIFILEINDEXTYPECOL	Specifies the position of the type information column. Initially, its value is 41.

You may control the information that is actually printed to the destination file through four variables:

MULTIFILEINDEXMAPFLG	Indicates that you want the file index to be printed to the destination file. Initially, it has the value T.
MULTIFILEINDEXFILESFLG	Indicates that you want the files output to the destination file. Initially, it has the value T.
MULTIFILEINDEXFNSMSFLG	Indicates that you want Masterscope information produced about each function in the destination file. Initially, it has the value NIL.
MULTIFILEINDEXVARSMSFLG	Indicates that all the variables used in the files will have some information about them produced in the report. The list of variables considered is determined by the answer to the Masterscope question "WHO IS USED BY ANY AND WHO IS SET BY ANY". The variable map is sorted by variable name independent of case. Initially, it has the value NIL.

The latter two variables use the interface to Masterscope included within MULTIFILEINDEX to obtain the necessary information. If either variable has the value T, MULTIFILEINDEX assumes that the source file(s) have been analyzed by Masterscope.

The compiled code for MULTIFILEINDEX resides in the file MULTIFILEINDEX.COM.

24.1.3 Implementing the ALL File Package Type

The ALL File Package type allows you to intermix definitions of various types for a given atom in a file. This capability is useful if you must produce listings since all the information is collected in a single location. The file defines both a File Package command and a File Package type.

The File Package command has the form

```
(ALL ⟨filepackagetypes⟩ ⟨specification1⟩ ...
⟨specificationN⟩)
```

The field ⟨filepackagetypes⟩ is a list of the file package types that may be stored in the list.

The fields ⟨specification1⟩ ... ⟨specificationN⟩ are lists consisting of an atom name and one or more types for which information is to be dumped.

Consider the following example:

```
(ALL
     (COMMENTS FNS VARS MACROS RECORDS)
     (GET.TREE FNS VARS)
     (GET.FOREST VARS
     (GET.BRANCH MACROS))
```

which dumps the function definition and the value of GET.TREE, the value of GET.FOREST, and the macro definition of GET.BRANCH.

The special form (ALL * ⟨file⟩ALL) is also recognized and handled as described in Section 17.2.20.

In the specifications, the type may also be a list of the form (PROPS ⟨property1⟩ ... ⟨propertyN⟩) which means to store the specified properties for that atom in the file.

When you store a definition for an atom A1 of type T1 on a file, the File Package looks for a command in which to put it (in the source file).

When an ALL command is encountered, the File Package uses the following procedure:

1. If T1 is not in the type list for the ALL command, then the definition will not be included in the command (e.g., ALL "overrides" the normal operation).
2. If A1 already exists in the command, then T1 will be inserted while attempting to maintain the order specified in the type list.
3. Otherwise, an entry of the form (A1 T1) is inserted in the command list.

4. If the File Package cannot put the definition into an existing command, it creates a new command.
5. If the type is a member of the list ALL-TYPES, then the File Package will create a command of the form (ALL ⟨all-types⟩ (A1 T1)).

The variable ALL-TYPES is a list of all the File Package types that will be intercepted by an ALL command. It is also the list of types that will be inserted into a newly created ALL File Package command as described above. Its initial value is

```
(COMMENTS FNS MACROS VARS LISPXMACROS USERMACROS)
```

but you may modify it to suit your own requirements.

24.1.4 Editing a File's History

EDITHIST provides extensions to the File Package that allow you to permanently preserve the history of new versions of files. It records a "who-edited-last" comment within the file for each function.

Every time a file is remade, EDITHIST will cause a new entry to be recorded in the file's edit history. The entry has the form

```
(⟨date⟩ ⟨author⟩ ⟨file⟩ (⟨changes⟩) (⟨comments⟩))
```

where

⟨date⟩	is the value of (DATE) when the file was remade
⟨author⟩	is the value of (OR INITIALS USERID)
⟨file⟩	is the full file name of the output
⟨changes⟩	is the same list of changed items that appears on the FILEDATES property
⟨comments⟩	is acquired (optionally) by ASKUSER during the execution of MAKEFILE.

When EDITHIST is loaded (usually during evaluation of INIT.LSP), it assigns default values to certain control parameters: ASKEDITHIST and LIMITEDITHIST. These variables are interpreted as follows:

ASKEDITHIST	Controls the invocation of ASKUSER when a file is remade. It takes one of the following values:

	NIL	Don't ask the user for comments and do not extend the edithistory.
	NOCOMMENT	Don't ask the user for comments, but do extend the edithistory with no comments field.
	COMMENT	Ask the user for comments and extend the edithistory with a comments field.
	T	Add an edithistory to any file that does not currently have it; ask the user for comments and extend the edithistory. This is the default value.
LIMITEDITHIST	Controls the pruning of the edithistory. It takes the form (#softlimit . #hardlimit) with a default value of (10 . 30). #softlimit specifies the number of recent entries to remain inviolable. #hardlimit specifies the threshold when pruning will be invoked.	

The edit history may grow quite large if the file is edited repeatedly during the development and debugging of a program. When a threshold is reached, the edit history will be pruned to reduce its size while preserving as much information as possible. EDITHIST uses the following procedure to prune the edit history:

1. Preserve the first N entries, where N is given by #softlimit.
2. Among the remaining entries, merge entries with the same author and no comments.
3. If the number of entries remains above the threshold, merge entries having no comments, but different authors (i.e., ignore the authors).

4. If still above the threshold, merge N of the oldest entries into a single entry that brings the number of entries below the threshold.

When entries are merged, the ⟨date⟩, ⟨author⟩, and ⟨file⟩ fields are CONSed together in the result. The ⟨changes⟩ list in the merged entry is the union of the individual changes in the contributing entries. The ⟨comments⟩ field in the merged entry is the concatenation of all the comments of the contributing entries.

At some point, you must manually edit the edithistory to remove the oldest information. EDITHIST will not discard information from entries, but merely create successive mergers that become unmanageable.

EDITHIST also defines additional options for MAKEFILE's OPTIONS field which provide temporary overrides:

ASK COMMENT	Temporarily resets ASKEDITHIST to T so you will be asked for a comment this one time only.
NOASK NOCOMMENT	Temporarily resets ASKEDITHIST to NOCOMMENT so you will not be asked for comments this one time only.

EDITHIST maintains the edithistory for each file in ALIST format keyed by the file name. The root of the list is EDITHISTALIST. You may examine or modify an edithistory using either (EDITDEF ⟨filename⟩ 'EDITHIST) or (SHOWDEF ⟨filename⟩ 'EDITHIST).

An edithistory is declared DONTCOPY so it will not be included in compiled files. If the compiled file is remade, the edithistory is recovered from the source file.

You may include a File Package command (EDITHIST ⟨filenames⟩) in the file's command list. If multiple file names are present (due to merged files), only the first will be subsequently updated.

24.1.5 Making Files Permanently Open

PERMSTATUS is intended to be used with the WHENCLOSE Package (see Section 16.5.3) to preserve a file's open status after an environment is restored from a SYSOUT file. The effect is to make the file "permanently" open.

You may achieve this effect (after loading the file PERMSTATUS.COM) by executing the expression

```
(WHENCLOSE ⟨filename⟩ 'STATUS 'PERMSTATUS)
```

after the file has been opened. Information will be saved about the file that permits its restoration after reloading a SYSOUT file. This information includes its access mode, file pointer value, and byte size.

The permanence of a file is not guaranteed. You may delete or rename the file or change its contents by means external to the INTERLISP environment. If a file cannot be found or restored during the loading of a sysout, a warning message will be printed.

Permanent files will be closed by CLOSEF or CLOSEALL or due to closing on end-of-file errors unless the appropriate attributes are specified via the WHENCLOSE Package.

24.2 EXTENSIONS TO MASTERSCOPE

Two LISP User's Packages extend the capabilities provided by Masterscope (see Chapter 26).

24.2.1 Dumping Masterscope's Knowledge

Masterscope builds a database describing one or more files which it has analyzed. The TELL Package allows you to obtain a complete dump of all of Masterscope's knowledge about a selected set of atoms, functions, or Masterscope relations. TELL defines a number of functions that allow you to extract varying amounts of information from the Masterscope database.

TELLABOUT allows you to obtain information about a list of atoms described in the Masterscope database. It takes the form

```
Function:       TELLABOUT

# Arguments:    4

Arguments:      1) a subject list, SUBJECTS
                2) a list of related atoms, UNIVERSE
                3) a question list, OMITQUESTIONS
                4) a destination file, DUMPFILE

Value:          The destination file name.
```

SUBJECTS is a list of atoms about which information is desired. UNIVERSE is a list of atoms which are related to the atoms appearing on SUBJECTS. Whenever one of these atoms is detected in a Masterscope relation, it will be printed.

OMITQUESTIONS is a list of question numbers (see below) for those questions which are not to be asked of the Masterscope database.

DUMPFILE is the destination file where the output from Masterscope will be written.

For each atom on SUBJECTS, TELL prints an entry that contains the answers to every question asked by ENUMERATEQUESTIONS of the Masterscope database except those identified by OMITQUESTIONS. The atom is substituted for the * in the questions described below. Members of UNIVERSE will

be substituted for WHO or WHOM in the questions. If the answer is true, then an entry is printed.

At the end of the file, TELLABOUT displays all of the questions asked and the value of UNIVERSE.

TELL assumes that Masterscope has already analyzed the functions which are the source of the information.

24.2.2 Enumerating the Masterscope Questions

ENUMERATEQUESTIONS prints a list of questions at your terminal that exhaustively describes all of the simple relations among objects that are indexed by Masterscope. Each questions is prefixed by an integer identifying the question which may be used in TELLABOUT. It takes the form

```
Function:       ENUMERATEQUESTIONS
# Arguments:    0
Arguments:      N/A
Value:          NIL, but the list of questions is a side
                effect.
```

The list of questions printed by ENUMERATEQUESTIONS at your terminal is as follows:

1. Whom does * bind?
2. Whom does * bind locally?
3. Whom does * bind as an argument?
4. Whom does * call?
5. Whom does * use?
6. Whom does * use freely?
7. Whom does * use locally?
8. Whom does * use globally?
9. Whom does * use as a record?
10. Whom does * use as a field?
11. Whom does * use as a property?
12. Whom does * use as a CLISP i.s.operator?
13. Whom does * set?
14. Whom does * set freely?
15. Whom does * set locally?
16. Whom does * set globally?
17. Whom does * smash?

18. Whom does * smash freely?
19. Whom does * smash locally?
20. Whom does * smash globally?
21. Whom does * test?
22. Whom does * test freely?
23. Whom does * test locally?
24. Whom does * test globally?
25. Whom does * reference?
26. Whom does * reference freely?
27. Whom does * reference locally?
28. Whom does * reference globally?
29. Who binds *?
30. Who binds * locally?
31. Who binds * as an argument?
32. Who calls *?
33. Who uses *?
34. Who uses * freely?
35. Who uses * locally?
36. Who uses * globally?
37. Who uses * as a record?
38. Who uses * as a field?
39. Who uses * as a property?
40. Who uses * as a CLISP I.S.operator?
41. Who sets *?
42. Who sets * freely?
43. Who sets * locally?
44. Who sets * globally?
45. Who smashes *?
46. Who smashes * freely?
47. Who smashes * locally?
48. Who smashes * globally?
49. Who tests *?
50. Who tests * freely?
51. Who tests * locally?
52. Who tests * globally?
53. Who references *?
54. Who references * freely?
55. Who references * locally?
56. Who references * globally?

24.2.3 Finding Out about Everything

You may find out about everything that Masterscope has seen by executing **EVERYONE**, which takes the form

```
Function:       EVERYONE
# Arguments:    0
Arguments:      N/A
Value:          A list of answers to the questions noted
                below.
```

EVERYONE returns a list which describes everything Masterscope knows about objects it has seen (i.e., that have been analyzed) except for system functions and I.S.OPRS. In particular, EVERYONE will answer the following questions:

1. Who is known?
2. Who is used?
3. Who is used as a property?
4. Who is used as a field?
5. Who is used as a record?

Dumping the Results of EVERYONE

You may dump the results of executing EVERYONE to a file by executing **TELLABOUTEVERYONE**, which takes the form

```
Function:       TELLABOUTEVERYONE
# Arguments:    2
Arguments:      1) a list of questions to be omitted,
                   OMITQUESTIONS
                2) a dump file name, DUMPFILE
Value:          The value returned by EVERYONE.
```

TELLABOUTEVERYONE calls TELLABOUT with the argument list: (EVERYONE() EVERYONE() DUMPFILE).

You may produce an exhaustive but not overly redundant set of information about functions in the Masterscope database using **BASISTELLABOUT**, which takes the form

```
Function:       BASISTELLABOUT
# Arguments:    1
```

Arguments:	1) a dump file name, DUMPFILE
Value:	The dump file name.

BASISTELLABOUT calls TELLABOUTEVERYONE with the argument list (REASONABLEOMISSIONS DUMPFILE).

REASONABLEOMISSIONS is a variable whose value is a list of the numbers of the questions whose answers are supersets of the answers to simpler questions. The only information that is not dumped by omission these questions appears to be information about I.S.OPRS.

These functions are stored in the file ⟨LISPUSERS⟩TELL.COM.

24.2.4 Automatic Masterscope Database Creation

Normally, you must manually create a Masterscope database by explicitly directing Masterscope to analyze a set of functions and index the information in the database. The functions stored in the DATABASEFNS package attempt to make this process automatic.

When DATABASEFNS is loaded, it modifies MAKEFILE, LOAD, and LOADFROM (see Chapter 17) to automatically update an existing Masterscope database.

When a file is made or loaded via MAKEFILE, LOAD, or LOADFROM, the Masterscope database will be automatically maintained if the atom corresponding to the file name has the property DATABASE with the value YES. Whenever a file is dumped via MAKEFILE, Masterscope will analyze any the functions in the file to determine if any have been modified or if new functions have been added to the file. A new database whose name is ⟨filename⟩.DATABASE will be written with information concerning the new or changed functions.

Whenever a file is loaded via LOAD or LOADFROM, the corresponding ⟨filename⟩.DATABASE file will also be loaded.

The database will be neither dumped nor loaded if the value of the property DATABASE is NO. If the file is loaded with LDFLG having the value SYSLOAD, then the value of the DATABASE property will be assumed to be NO.

If the value of the DATABASE property is neither YES nor NO (a good value is ?), then you will be asked whether or not you want the Masterscope database automatically maintained at the first execution of MAKEFILE, LOAD, or LOADFROM for the indicated file. Your answer is stored as the value of the DATABASE property so that you will not be repeatedly asked the question. This feature is controlled by two global variables:

SAVEDBFLG	A value of YES or NO will be stored instead of asking the question if the value of the DATABASE property is neither YES or NO depending on whether its value is YES or NO. Its value is initially ASK.

LOADDBFLG	A value of YES or NO will be stored instead of asking the question if the value of the DATABASE property is neither YES or NO depending on whether its value is YES or NO. Its value is initially ASK.

You will be queried if the value of either of these variables is ASK.

Dumping and Restoring MSDATABASE

You may explicitly dump and restore Masterscope databases using **SAVEDB** and **LOADDB**, which take the form

Function:	SAVEDB LOADDB
# Arguments:	1
Arguments:	1) a file name, FILE
Value:	The file name.

SAVEDB dumps a database for FILE and then sets the DATABASE property of the atom corresponding to the file name to YES. Thus, database maintenance for that file will be automatically performed on each subsequent MAKEFILE. The file that is created has the name ⟨filename⟩.DATABASE.

LOADDB loads the database file having the name ⟨filename⟩.DATABASE, if one exists, associated with FILE. After the database is loaded, LOADDB sets the value of the property DATABASE of the atom corresponding to the file name to YES to make database maintenance automatic thereafter.

You may not load a database file with the standard INTERLISP loading functions because of its special format. LOADDB is the only way to access a database file.

These functions are stored in the file ⟨LISPUSERS⟩DATABASEFNS.COM.

24.3 THE DECL PACKAGE

Several conventional languages allow you to restrict the scope of arguments to functions by specifying the datatypes of variables that are passed as arguments to those functions. Normally, this takes the form of declaring the variable name and typing in the argument list associated with the function (or procedure) definition. Examples of such languages include PASCAL and ADA.

INTERLISP normally allows you to pass any type of expression or variable to a function. It is incumbent upon the called function to determine the type of the argument and whether or not the type is a valid one for the function to oper-

ate upon. Thus, in many functions, if the user is a careful programmer, you will see a COND statement checking the types of the arguments.

The DECL Package provides a mechanism for introducing lexical scoping in INTERLISP functions. Lexical scoping constrains the behavior of INTERLISP functions such that the type of arguments passed to the function is checked when the function is called. To do so, the DECL Package modifies the actions of both the INTERLISP interpreter and the compiler.

Type declarations on arguments can make the function more readable since they indicate more clearly (if mnemonic names are chosen for functions) what the function is intended to do. They also facilitate debugging of functions since they assist in localizing errors that are the result of type incompatibilities between the arguments passed and those expected by the function. Type declarations will be recognized by Masterscope when the DECL Package is loaded.

24.4 TRANSOR: A LISP TRANSLATOR

As explained in Chapter 1, there are two major dialects of LISP. The other dialect, MACLISP, has a number of subdialects as described in Section 1.2. Differences between INTERLISP and these dialects are noted therein. TRANSOR is a LISP-to-LISP translator that allows you to effectively translate LISP programs in one dialect into another (i.e., the translation may be to or from INTERLISP).

TRANSOR is driven by a set of transformation specifications which describe the differences between the two dialects. The differences are expressed as INTERLISP Editor commands which will convert the input dialect form into an output dialect form. When TRANSOR is loaded along with the transformation file, it will read the input file and apply the edit transformations to all expressions within the file. It produces an output which should be suitable for loading on the target LISP system. In addition, TRANSOR will produce a set of transformation notes that describe the major changes made to the input source code to produce the output file. It also notes any additional changes that require further attention from the user.

TRANSOR may be thought of as a driver for a massive editing task. While the primary usage is to translate from one LISP dialect to another, proper specification of the edit transformations may allow you to translate a large percentage of a source file in another language (such as C) to INTERLISP or vice versa.

24.5 OTHER LISP USER'S PACKAGES

There are numerous other LISP User's Packages that are available under INTERLISP-10 which work only with INTERLISP-10. There are also many packages available with INTERLISP-D which will be described in Volume 2.

The following sections provide just a brief glimpse of some of the other packages available for INTERLISP-10. You may find documentation on these packages in the LISPUSERS directory on a DECSystem-10 computer system.

24.5.1 The Pattern Match Compiler

The Pattern Match Compiler is a package that provides additional flexibility in specifying patterns to be matched within CLISP statements.

In general, you want to answer the question: Does the expression X look like the pattern P? The Pattern Match Compiler provides a syntax that allows you to specify patterns and the rules for matching them.

24.5.2 The Hash File Package

The Hash File Package allows you to effectively extend the memory of the INTERLISP-10 system by permitting information associated with atoms or strings to be stored on a file rather than in memory. The information is retrieved as it is needed by the program. To make the retrieval efficient, the Hash File Package makes use of the INTERLISP-10 page mapping facility to access the information stored in the file(s).

24.5.3 EDITA: The Array Editor

EDITA is an Array Editor. However, its most frequent use is in editing compiled functions which are represented as arrays in INTERLISP-10. While EDITA is also available on INTERLISP-D, capabilties pertaining to the editing of compiled code will only work for INTERLISP-10.

In general, you invoke EDITA to edit a function. It then accepts commands similar to the DECSystem-10 DDT (Dynamic Debugging Tool). The commands allow you to open registers (i.e., memory locations), examine symbol tables, change the contents of memory locations, and search for specific values.

24.5.4 CJSYS: Access to the Operating System

CJSYS is a package running under INTERLISP-10 that allows you direct access to the underlying operating system. Operating system calls in TENEX and TOPS-20 are call JSYS (Jump to SYStem) traps; hence the name of the package.

24.5.5 EXEC: A TENEX Executive in INTERLISP

EXEC is a package that defines additional Programmer's Assistant commands that provide you with some of the capabilities of the TENEX EXEC. These capabilities provide you with access to certain operating system features while not requiring knowledge about the specific interfaces (i.e., no need to use CJSYS). It also defines functions that provide the same capabilities so that you may exercise them from within your programs.

24.5.6 The NET Package

The NET Package provides functions for establishing ARPANET connections from INTERLISP-10 jobs. You may open, close, and check connections with other computers in the ARPANET (provided you have an account on them).

24.5.7 FTP: The File Transfer Package

FTP is a package that allows you to transfer files between hosts on the ARPANET. When FTP is loaded, you may access files at other ARPANET hosts using the basic file system commands INFILE, OUTFILE, and OPENFILE, and you may read and write those files as if they were resident at your local host.

25

The Programmer's Assistant

The Programmer's Assistant is a subsystem that interacts with you to assist in creating and running programs. You may think of it as an aide who looks over your shoulder at what you are doing. Normally, expressions that you type in are executed without any interference by the Programmer's Assistant. However, it does recognize a set of commands that cause it to do things for you. Among the functions that it may perform are

1. Remembering and repeating a sequence of commands that you have previously entered.
2. Undoing the effects of commands you have recently executed.
3. Analyzing erroneous input and presenting you with a diagnosis of what went wrong.

The Programmer's Assistant is dispersed throughout INTERLISP. I have chosen to discuss it in two chapters in order to enhance the clarity of presentation. This chapter concentrates upon the main features and concepts that underly the Programmer's Assistant. Chapter 28 discusses the History Package which supports the auditing and undoing of expressions.

25.1 THE CONCEPT OF UNDOING

Undoing means to reverse the effects of an expression that you have previously typed in and INTERLISP has executed. Undoing requires that each expression that might be undone have information stored on a history list to enable reversal of its side effects. The Assistant has no way of knowing when it is about to perform a destructive operation. To constantly check for such occurrences would seriously degrade the efficiency of the system. Rather, the Assistant merely "watches" the execution of expressions. HISTORYSAVE (see Section 28.6.1) automatically saves undoable changes on the history list. Undoing involves read-

ing the history list to access the information necessary to perform the inverse operations.

As we said previously, the Assistant is relatively passive until you instruct it to do something. Only then does it check to see if the necessary information is available on the history list. If the user explicitly identifies the event and nothing has been saved, the Assistant prints the message "NOTHING SAVED". Otherwise, it searches for the last undoable event (ignoring events already undone as well as other UNDO operations) and undoes it.

The primary question is: How does the system know what is undoable and what is not? Underlying this question is the problem of efficiency. The ability to undo every operation would rapidly consume storage to save the information necessary to perform the inverse operation. Moreover, only a few system functions are really destructive. Thus, these are the only ones we need to know something about in order to undo them. Perversely, however, these are the functions that are executed most often, which leads us back to the problem of storage consumption.

The Assistant has solved this problem by forcing two constraints on undoing:

1. Each function which may be undone must save the appropriate information necessary to undo its effects on the history list.
2. For each primitive destructive function, there are actually two implementations—one that is undoable and one that is not. The undoable function saves the necessary information on the history list when it is executed. You decide what may be undone by your choice of the implementation of the destructive function.

Most of the INTERLISP subsystems use the undoable versions of the primitive destructive functions. You may also use these functions if you want functions that you execute to be undoable as well.

The Assistant makes an assumption that all expressions that are typed into LISPX should be undoable because errors are most likely to occur during interactive dialogue. Thus, the Assistant substitutes the corresponding undoable function for each destructive operation that it encounters on type-in. This assumption is based on the fact that most expressions that are typed in rarely involve iterations or lengthy operations directly (when was the last time that you typed a PROG statement upon receiving a prompt?).

LISPX (see Section 25.2) scans all input statements before executing them. It substitutes undoable versions of the primitive destructive functions for all regular versions detected in the input expression. You may wish to make other functions undoable by performing your own substitutions. You may do so by defining LISPXUSERFN (see Section 25.3) to be a function that processes your input expressions according to your own requirements.

Undoing may be fraught with danger. Certain sequences of operations must be performed in exactly the reverse order for the system to revert to its proper

former state. These include functions that make substantial use of /RPLACA and /RPLACD or that perform several successive operations on a single list. Good programming practice requires that you undo these types of functions in the reverse order of their execution.

25.2 LISPX: TYPE-IN EVALUATION

The Assistant uses LISPX (for LISP eXecutive) to obtain an expression for evaluation. INTERLISP reads the first expression typed into the system after the prompt character. It calls LISPX with this expression assigned to the variable LISPXX. **LISPX** takes the form

```
Function:       LISPX

# Arguments:    5

Arguments:      1) the initial expression, LISPXX
                2) the value of the prompt character,
                   LISPXID
                3) lispx macros, LISPXXMACROS
                4) a user processing function,
                   LISPXXUSERFN
                5) a flag, LISPXFLAG

Value:          The value of the expression as computed.
```

If the value of LISPXXUSERFN is non-NIL, the input expression will be passed to it for processing. If so, you do not need to define a value for LISPXUSERFN.

You may invoke LISPX from within your program if you wish certain inputs to be undoable. LISPXID allows you to define your own prompt character to signify that input is being directed to your own function rather than INTERLISP.

LISPX processes the input as follows:

1. If the value of LISPXX is not a list, LISPX calls READLINE to read the remaining expressions up to a break character or a carriage return. The value of LISPXX is concatenated with the value of READLINE and the result is stored on the history list. LISPX then decides what to do with the input it has received. The possibilities include

    a. It is a history command to be executed. LISPX invokes HISTORYSAVE, executes the command, and returns the value of HISTORYSAVE.
    b. It is a LISPXMACRO (see Section 25.2.1).
    c. It is to be processed by a LISPXUSERFN (see Section 25.3).
    d. It calls EVAL or APPLY depending upon the input form.

2. If LISPXX is a list whose CAR is LAMBDA or NLAMBDA, LISPX invokes LISPXREAD to obtain the arguments which include the argument list and all expressions composing the body of the function. A list is indicated by an opening (.
3. If LISPXX has the value) or], READLINE returns (NIL) instead of NIL. This permits you to call a function with no arguments as in

```
←CLEANUP]
```

which then uses FILESLST to do its work. Note that this distinguishes the execution of the function CLEANUP from the variable CLEANUP which would have been typed as

```
←CLEANUP⟨CR⟩
```

4. If the expression consists of a list preceded by other characters than spaces and terminated by a) or a], LISPX treats the expression as being in APPLY format and proceeds accordingly.

```
←EDITF(REDRAW.MAP)
```

which invokes the Editor on the function REDRAW.MAP.

LISPX saves the inputs for all but the history commands on a history list before executing. Thus, even if the operation is aborted, you may redo it, fix it, etc.

If the input is an EVAL or APPLY form or a form without parentheses (e.g., a CLISP statement), the function name will be looked up in LISPXFNS. If you want a different function called than you typed in, place an entry of the form (⟨fn1⟩ . ⟨fn2⟩) on LISPXFNS which will cause LISPX to invoke ⟨fn2⟩ whenever it detects ⟨fn1⟩ in a form.

For history commands, LISPX merely unreads the information from the history list and exits. However, if your program calls LISPX, history commands will not work unless your program obtains input via LISPXREAD. Moreover, your program must save the inputs that it reads on the history list in the approved format.

LISPX will perform spelling correction using LISPXCOMS as its spelling list. LISPXCOMS is a list of commands recognized by LISPX. Spelling correction is applied whenever LISPX receives an atom which is unbound or a function which is undefined.

25.2.1 A Definition for LISPX

We might define LISPX as follows:

```
(DEFINEQ
    (lispx (lispxx lispxid lispxxmacros lispxxuserfn)
        (*
            If the caller provides no special macros,
            use the standard ones defined by the
            Assistant.
        )
        (AND
            (NULL lispxxmacros)
            (SETQ lispxxmacros LISPXMACROS))
        (*
            If the caller provides no user function,
            or LISPXUSERFN has the value NIL, use the
            standard LISPXUSERFN.
        )
        (OR
            lispxxuserfn
            (NULL LISPXUSERFN)
            (SETQQ lispxxuserfn LISPXUSERFN))
        (PROG (lispxop lispxlistflg lispxline lispxhist
               lispxvalue lispxtemp lispxoptmp)
              (COND
                  ((NULL lispxx)
                      (*
                          A spurious right bracket or
                          parenthesis was detected.
                          Print NIL and try again.
                      )
                      (RETURN
                          (AND
                              (NOT silence)
                              (PRINT NIL T))))
                  ((NLISTP lispxx)
                     (SETQ lispxline
                           (READLINE NIL T))))
        top
              (COND
                  ((LISTP lispxx)
                      (*
                          Break out the operator and
                          its arguments. LISPXOP is
                          always the name of the
                          history command and
                          LISPXLINE is its arguments.
                      )
```

```
                    (SETQ lispxop (CAR lispxx))
                    (SETQ lispxline (CDR lispxx))
                    (SETQ lispxlistflg T))
               ((NOT (LITATOM lispxx))
                    (*
                         If it is not a litatom,
                         then it is not a command.
                         It could be an error in
                         typing or a number.
                    )
                    (GO notcommand)
               (T
                    (SETQ lispxop lispxx)))
select
     (SELECTQ lispxop
          ((RETRY REDO FIX USE ...)
               (GO redocommand))
          (NAME
               (COND
                    ((NULL lispxline)
                         (*
                              NAME may also be
                              the name of a
                              variable defined
                              by the user, so
                              try not to
                              confuse the two.
                         )
                         (SETQ lispxx 'E)
                         (SETQ lispxline (LIST
                         'NAME))
                         (GO execute.command)))
               (GO redocommand))
          (UNDO
               (AND
                    (SETQ lispxhist
                          (HISTORYSAVE
                          LISPXHISTORY
                                        lispxid
                                        NIL
                                        'UNDO
                                        lispxline))
                    (RPLACA lispxhist
                            (UNDOLISPX
                            lispxline))))
```

```
            (?
                 (PRIN1 "Commands are:" T)
                 (PRINT LISPXCOMS T))
            ((RETRY: FORGET ??)
                 (*
                      If REREADFLG is ABORT, then
                      do a CTRL-B via program
                      call.
                 )
                 (AND
                      (EQ REREADFLG 'ABORT)
                      (ERRORB))
                 (COND
                    ((EQ lispxop 'RETRY:)
                      (*
                           After the RETRY comes
                           the command with
                           arguments to be
                           retried.
                      )
                      (SETQ lispxx (CAR
                      lispxline))
                      (SETQ lispxline (CDR
                      lispxline))
                      (GO top)))
                    ((EQ lispxop 'FORGET)
                           (lispx-forget)
                           (PRINT 'FORGOTTEN T))
                    ((EQ lispxop '??)
                           (lispx-??))))
            (GO notcommand))
       (RETURN '?)
notcommand
       (*
            Determine if LISPXOP is a lispxmacro
            either predefined by INTERLISP or
            specified by the user in this call to
            LISPX.
       )
       (SETQ lispxoptmp
             (ASSOC lispxop
                    (OR lispxxmacros
                        (CAR (QUOTE
                        LISPXMACROS)))))
       (COND
```

```
(lispxoptmp
     (AND
          lispxlistflg
          (SETQ lispxline NIL))
     (GO execute.command))
((MEMBER lispxop LISPXCOMS)
     (*
          At this point, LISPXOP is
          neither a built-in command
          nor a macro. However, the
          user may have put it on
          LISPXCOMS in order to
          process it by LISPXUSERFN.
     )
     (AND
          lispxlistflg
          (SETQ lispxline NIL))
     (GO execute.command))
((SETQ lispxoptmp
       (ASSOC lispxop
       lispxhistorymacros))
     (*
          If the command is actually
          a macro, find its
          definition and redo it.
     )
     (GO redocommand))
(lispxlistflg
     (*
          The input is a list.
     )
     (COND
          ((EQ (CAR lispxx) 'LAMBDA)
               (*
                    A function
                    definition!
               )
               (SETQ lispxline
                     (LIST
                          (LISPXREAD
                          T))))
          (T
             (AND
               (LITATOM (CAR lispxx))
               (SETQ lispxop (CAR
               lispxx))
```

```
                                (SETQ lispxline (CDR
                                lispxx))
                                (GO select))))
                        (GO execute.command))
                    ((NULL lispxline)
                       (AND
                         (LITATOM lispxx)
                         (COND
                            ((NEQ (CAR lispxx) 'NOBIND)
                                  (ADDSPELL lispxx 3))
                            ((NEQ (EVALV lispxx)
                            'NOBIND))
                            ((SETQ lispxop
                                  (FIXSPELL lispxx
                                            70
                                            LISPXCOMS
                                            NIL
                                            T))
                                (COND
                                   ((LISTP lispxop)
                                      (SETQ lispxline
                                            (LIST
                                               (CDR
                                               lispxop)))
                                      (SETQ lispxop
                                            (CAR
                                            lispxop))))
                                (SETQ lispxx lispxop)
                                (GO select))))
                         (GO execute.command)))))
execute.command
     (RETURN (LISPXDO-IT))
redocommand
     (LISPXREDOCOMMAND)
     (RETURN '?)))
))
```

25.2.2 LISPX Macros

You may define your own LISPX commands and place them on a list which is the value of LISPXMACROS. LISPXMACROS is a list of elements of the form

(⟨command⟩ ⟨definition⟩)

Whenever one of the ⟨command⟩s is the CAR of an expression read by LISPX, LISPXLINE will be bound to the remainder of the input. The input

event is recorded on the history list, and ⟨definition⟩ is evaluated. Its value will be stored as the value of the event.

An alternative form may appear as

```
(⟨command⟩ NIL ⟨definition⟩)
```

which means do not store the event on the history list.

Initial Definition of LISPXMACROS

LISPXMACROS is initially defined as

```
   ((NDIR
(DODIR LISPXLINE '(PP COLUMNS 17) '* 0))
   (DEL
       (DODIR LISPXLINE 'DELETE " " 'L))
   (CONN
       (CNDIR (CAR LISPXLINE)
              (AND
                     (LISTP (CDR LISPXLINE))
                     (PROG1
                            (CADR LISPXLINE)
                            (RPLACA (CDR
                            LISPXLINE))))))
   (REMEMBER
       (REMEMBER LISPXLINE))
   (REMEMBER:
       (PROG1
            (RESETVARS (FILEPKGFLG)
                       (RETURN
                           (EVAL (LISPX/ (CAR LISPXLINE)
                                 LISPXID)))
            (MARKASCHANGED (CAR LISPXLINE)
            'EXPRESSIONS)))
   (OK
       (RETFROM
            (OR (STKPOS 'USEREXEC) 'LISPX)
            T T))
   (AFTER
       (LISPXSTATE (CAR LISPXLINE) 'AFTER))
   (BEFORE
       (LISPXSTATE (CAR LISPXLINE) 'BEFORE))
   (RETRIEVE
       (PROG (X REREADFLG)
             (SETQ X (GETP (CAR LISPXLINE) '*HISTORY*))
             (COND
```

```
                ((NULL X)
                    (ERROR (CAR LISPXLINE) "?" T)))
            (MAPC (CDDR X)
                (FUNCTION
                    (LAMBDA (X)
                        (HISTORYSAVE LISPXHISTORY
                        X))))
            (RETURN (CAR LISPXLINE))))
(SHH
    NIL
    (COND
        ((OR
            (CDR (LISTP LISPXLINE))
            (AND
                (MEMBER (LASTC T) '(%) %]))
                (LITATOM (CAR LISPXLINE))))
            (APPLY (CAR LISPXLINE)
                (COND
                    ((AND
                        (LISTP (CADR LISPXLINE))
                        (NULL (CDDR LISPXLINE)))
                        (CADR LISPXLINE))
                    (T
                        (CDR LISPXLINE)))))
        (T
            (EVAL
                (COND
                    (LISPXLINE
                        (CAR LISPXLINE)
                    (T 'SHH))))))
(DIR
    (DODIR LISPXLINE)))
```

25.2.3 User Processing of Input

You may intercept the processing of input from the terminal by setting LISPXUSERFN to T. You must also define LISPXUSERFN with the definition of the function that will process the input.

When LISPXUSERFN is T, its function definition will be applied to all inputs that are not recognized as Programmer Assistant commands or that appear on LISPXMACROS. LISXPUSERFN should attempt to do some recognition of the input. The input will already be recorded as an event on the history list. Thus, LISPXUSERFN should set LISPXVALUE to the value for the event and return T. LISPXVALUE will be stored as the value of the event and will be printed at your terminal.

If LISPXUSERFN returns NIL, then EVAL or APPLY is called in the usual way as appropriate to the input.

LISPXUSERFN is usually used to enhance user executives (see Section 25.3). LISPXUSERFN may be used, when appropriately defined, to process natural language input entered by a user at type-in.

The function definition for LISPXUSERFN is a function of two arguments: X and LINE. X is the first expression typed in, and LINE is the remainder of the input line as read by READLINE (see Section 14.2.4). A general definition for a LISPXUSERFN might appears as follows:

```
(DEFINEQ
  (lispxuserfn (x line)
     (PROG (command)
           (COND
              ((AND (NULL line) (LISTP x))
                    (SETQ command (recognize (CAR x)))
                    (SETQ LISPXVALUE
                          (EVAL command (CDR x))))
              ((AND (ATOM x) (LISTP line))
                    (SETQ command (recognize x))
                    (SETQ LISPXVALUE
                          (EVAL command line)))
              ((AND (ATOM x) (LISTP (CAR line)))
                    (SETQ command (recognize x))
                    (SETQ LISPXVALUE
                          (EVAL command (CAR line)))))
           (RETURN LISPXVALUE))
  ))
```

which handles the three forms of input as follows:

1. If you typed

   ```
   ←(command argument1 argument2 ... argumentN)
   ```

   then X is (command argument1 ... argumentN) and LINE is NIL:

   ```
   ←(SETQ X 100)
   ```

2. If you typed

   ```
   ←command argument1 argument2 ... argumentN
   ```

then X is command, as in EVAL format, and LINE is (argument1 ... argumentN).

```
←USE SET FOR SETQ
```

3. If you typed

```
←command(argument1 argument2 ... argumentN)
```

then X is command, as in APPLY format, and LINE is ((argument1 ... argumentN)).

```
←PP(REAL)
(REAL
     (LAMBDA (CX)
          (RECORDACCESS (QUOTE REAL)
                        CX NIL (QUOTE FETCH))))
```

25.3 ESTABLISHING A USER EXECUTIVE

Within your program, you may want to establish your own executive routine that interprets a set of commands defined by your application. **USEREXEC** allows you to define your own prompt character, macros, and scanning function. It calls LISPX iteratively until it encounters the lispxmacro OK or via a RETFROM. It takes the format

```
Function:       USEREXEC

# Arguments:    3

Arguments:      1) A prompt character, LISPXID
                2) lispxmacros, LISPXXMACROS
                3) a user scanning function, LISPXXUSERFN

Value:          The value of the last command.
```

We might define USEREXEC as follows:

```
(DEFINEQ
     (userexec (lispxid lispxxmacros lispxxuserfn)
          (PROG (readbuf)
               (*
                   READBUF is used to store the
                   remainder of the input line after
                   execution of READLINE.
               )
```

```
                (*
                    If you do not define your own prompt
                    character, then the default is used.
                )
                (AND
                    (NULL lispxid)
                    (SETQ lispxid '←))
        loop
                (*
                    Print the prompt character to the
                    terminal.
                )
                (PROMPTCHAR lispxid T LISPXHISTORY)
                (*
                    Read and evaluate the input with
                    proper protection.
                )
                (ERSETQ
                    (LISPX (LISPXREAD T)
                           lispxid
                           lispxxmacros
                           lispxxuserfn))
                (GO loop))
))
```

If you do not provide your own prompt character, USEREXEC assumes the standard system prompt character. Note that the call to LISPX is errorset protected to prevent an error from occurring due to erroneous input.

25.4 UNDOABLE VERSIONS OF DESTRUCTIVE FUNCTIONS

As mentioned above, each primitive destructive function has a corresponding version which saves information on the history list so that the effects of that function may be undone. LISPX substitutes the undoable version for the normal version on any statements that are typed in. You can make your functions undoable by using these undoable versions in your own functions.

25.4.1 Undoable Sets

SETs or SETQs may be made undoable by calling **SAVESET** instead of SET or SETQ. SAVESET stores the necessary information on the history list to undo its effects. It saves the old value of the variable (other than NOBIND) on the property list of the atom under the property VALUE. It also prints the message (⟨variable⟩ RESET). SAVESET has the following format

```
Function:       SAVESET

# Arguments:    4

Arguments:      1) a variable name, NAME
                2) a new value, NEWVALUE
                3) a top-level flag, TOPFLG
                4) a save flag, SAVEFLG

Value:          The new value of the variable, NEWVALUE.
```

Consider the following examples:

```
←MAP.WIDTH
NOBIND

←(SETQ MAP.WIDTH 512)
512

←(GETPROP MAP.WIDTH 'VALUE)
NIL

←(SETQ MAP.WIDTH 128)
(MAP.WIDTH RESET)
128

←(GETPROP MAP.WIDTH 'VALUE)
512
```

Because SAVESET is substituted for SET or SETQ by LISPX, it will also add any variables passed to it to the spelling list, SPELLINGS3.

SAVESET also calls the File Package to update the appropriate file records when any variable is changed that was set as the result of loading a file. MARKASCHANGED is invoked to ensure that the File Package notices that the file must be rewritten.

If DFNFLG (see Section 8.2.4) is not T, SAVESET prints the message (⟨name⟩ RESET). The old value is stored under the property VALUE on NAME's property list.

If TOPFLG is T, SAVESET changes the contents of the variable's value cell rather than scanning the pushdown list. If DFNFLG is ALLPROP as well, then NEWVALUE is stored as the value of the property VALUE. This allows you to load files without disturbing the current values of variables.

If SAVEFLG is NOSAVE, then the old value is not stored on the property list. Moreover, NAME is not added to the spelling list. However, the effects are still undoable because the relevant information has been stored in the history list.

```
←(SAVESET 'MAP.WIDTH 1024 NIL 'NOSAVE)
1024
```

```
←(GETPROPLIST 'MAP.WIDTH)
NIL
```

If SAVEFLG has the value NOPRINT, the old value is saved, but the message is not printed.

A Definition for SAVESET

We might define SAVESET as follows:

```
(DEFINEQ
  (saveset (name newvalue topflg saveflg)
     (COND
          ((NOT (LITATOM name))
               (*
                    If NAME is not an atom, generate an
                    error.
               )
               (ERROR "ARG NOT LITATOM" name))
          ((NULL name)
               (ERROR "ATTEMPT TO SET NIL" newvalue)))
     (PROG (pointer oldvalue temporary)
           (*
               Set POINTER to atom or stack frame of last
               binding.
           )
           (SETQ pointer
                 (COND
                    (topflg name)
                    (T
                         (*
                              Scan the stack for the last
                              binding of the variable.
                         )
                         (STKSCAN name
                                  (STKARG 0 (STKPOS
                                  'SAVESET))))))
           (SETQ oldvalue (CAR pointer))
           (COND
               ((AND topflg
                     (EQ DFNFLG 'ALLPROP)
                     (NEQ (CAR pointer) 'NOBIND))
                     (*
                         Save the new value under the
                         property VALUE on the atom's
                         property list.
                     )
```

```
                (/PUT name 'VALUE newvalue)
                (RETURN newvalue))
           ((OR
                (NOT (ATOM pointer))
                (EQ saveflg 'NOSAVE))
                (*
                     Check either for top-level
                     binding or for a call from /SET
                     or /SETQ.
                )
                (GO getout))
           ((OR
                (EQ oldvalue 'NOBIND)
                (EQUAL oldvalue newvalue)))
           (T
                (COND
                  ((NEQ DFNFLG T)
                     (SETQ temporary (CONS name
                     '(RESET)))
                     (AND
                        (NEQ saveflg 'NOPRINT)
                        (NULL
                          (SOME
                               (GET LISPXHIST
                               'LISPXPRINT)
                               (FUNCTION
                                  (LAMBDA (X)
                                     (AND
                                       (EQ (CAR X)
                                       'PRINT)
                                       (EQUAL (CADR
                                       X)
                                       temporary))))))
                          (LISPXPRINT temporary T))
                     (/PUTPROP name 'VALUE
                     oldvalue)))))
      (*
          ADD NAME to USERWORDS since it was typed
          in.
      )
      (AND
          DWIMFLG
          (ADDSPELL name T))
getout
      (AND
          (LISPXHIST
```

```
                (UNDOSAVE
                    (LIST 'UNDOSET pointer name oldvalue)
                    LISPXHIST))
                (*
                    Set the value of NAME here, finally.
                )
                    (RPLACA pointer newvalue)
                    (RETURN newvalue))
    ))
```

Unsetting Values

You may restore the old value by invoking **UNSET**, which takes the form

```
Function:       UNSET
# Arguments:    1
Arguments:      1) the name of an atom, NAME
Value:          The value of NAME.
```

UNSET saves the current value under the property VALUE and sets the value of the variable to the contents of VALUE. It does not print any messages. You may use UNSET to alternate between two variable values. This is particularly useful in testing programs. Consider the following example:

```
←(SAVESET 'MAP.WIDTH 512)
512
←MAP.WIDTH
512
←(GETPROPLIST 'MAP.WIDTH)
(VALUE 1024)
←(UNSET 'MAP.WIDTH)
MAP.WIDTH
←MAP.WIDTH
1024
←(GETPROPLIST 'MAP.WIDTH)
(VALUE 512)
```

UNSET is called with a single argument, the NAME of the variable whose old value is to be restored. UNSET merely invokes SAVESET with the proper arguments.

A Definition for UNSET

We might define UNSET as follows:

```
(DEFINEQ
    (unset (name)
        (SAVESET name
                 (GETPROP name 'VALUE)
                 T
                 NOPRINT)
    ))
```

Alternative Forms

Alternative forms for SETQ and SETQQ may be defined in terms of SAVESET.

```
(DEFINEQ
    (savesetq
        (NLAMBDA (name newvalue)
            (SAVESET name (EVAL newvalue))
    )))
(DEFINEQ
    (savsetqq
        (NLAMBDA (name newvalue)
            (SAVESET name newvalue)
    )))
```

Alternative Forms for RPAQ and RPAQQ

Note also that RPAQ and RPAQQ (see Section 3.9) operate like SETQ and SETQQ except that they bind the top-level value of the variable. We can define them using SAVESET as follows:

```
(DEFINEQ
    (rpaq
        (NLAMBDA (name newvalue)
            (SAVESET name (EVAL newvalue) T)
    )))
(DEFINEQ
    (rpaqq
        (NLAMBDA (name newvalue)
            (SAVESET name newvalue T)
    )))
```

25.4.2 Replacing the Top-Level Value

Both RPLACA and RPLACD affect the top-level value of a variable. Their undoable versions are /RPLACA and /RPLACD. They may be defined as

```
(DEFINEQ
  (/rplaca (name newvalue)
```

```
        (AND LISPXHIST
             (UNDOSAVE
                (COND
                    ((LISTP name)
                        (CONS name
                              (CONS (CAR name)
                                    (CDR name))))
                    (T
                        (LIST '/RPLACA
                              name
                              (CAR name))))
                LISPXHIST))
        (RPLACA name newvalue)
    ))
(DEFINEQ
    (/rplacd (name newvalue)
        (AND LISPXHIST
             (UNDOSAVE
                (COND
                    ((LISTP name)
                        (CONS name
                              (CONS (CAR name)
                                    (CDR name))))
                    (T
                        (LIST '/rplacd
                              name
                              (CDR name)))
                LISPXHIST))
        (RPLACD name newvalue)
    ))
```

Note that /RPLACA and /RPLACD have the same effect as RPLACA and RPLACD. Moreover, if no history list is defined (i.e., the History Package is not being used), they are exactly equivalent.

25.4.3 Undoing Mapping Functions

Mapping functions (see Chapter 12) apply a function to one or more arguments sequentially. Side effects of the function or functions called by it may alter the value or structure of the individual arguments. We can define an undoable version of a mapping function which saves the necessary information on the history list by using the undoable versions of primitive destructive functions. As an example, consider the definition for an undoable version of MAPCONC that is called /MAPCONC:

```
(DEFINEQ
   (/mapconc (mapx mapfunction.1 mapfunction.2)
       (PROG (map.lst map.entry map.result)
        loop
             (COND
                ((NLISTP mapx)
                   (RETURN map.lst))
                ((SAVESETQ map.result
                           (APPLY* mapfunction.1
                                   (CAR mapx)))
                   (COND
                      (map.entry
                         (/rplacd map.entry map.result))
                      (T
                         (SAVESETQ map.lst
                                   (SETQ map.entry
                                         map.result))))
                   (PROG NIL
                    loop2
                         (COND
                            ((SAVESETQ map.result
                                       (CDR map.entry))
                               (SAVESETQ map.entry
                                         map.result)
                               (GO loop2)))))
                (SAVESETQ mapx
                          (COND
                             (mapfunction.2
                                  (APPLY* mapfunction.2
                                          mapx))
                             (T
                                  (CDR mapx))))
                (GO loop))
     ))
```

Similar definitions may be constructed for each of the other mapping functions.

25.4.4 Undoing Function Definitions

Function definitions may be stored on the property list of an atom under the property EXPR. Two functions may set these definitions: PUTD and MOVD. We can construct undoable versions of these functions as follows:

```
(DEFINEQ
   (/putd (a.function a.definition)
```

```
    (PROG (temporary)
          (SETQ temporary (GETD a.function))
          (COND
              ((NOT (ATOM a.function))
                   (RETURN NIL)))
          (PUTD a.function a.definition)
          (AND LISPXHIST
               (UNDOSAVE
                   (LIST '/putd a.function temporary)
                   LISPXHIST))
          (RETURN a.definition))
))
```

Note that we test A.FUNCTION to make sure it is atomic so that no error results when we attempt to do the PUTD.

Using /PUTD, we can now define /MOVD:

```
(DEFINEQ
   (/movd (function.1 function.2 flag)
     (PROG (newflag)
           (SETQ newflag (NULL (GETD function.2)))
           (/PUTD function.2
                  (COND
                     (flag
                         (COPY (GETD function.1)))
                     (T
                         (GETD function.1))))
           (AND FILEPKGFLG
                (EXPRP function.2)
                (NEWFILE? function.2 NIL newflag))
           (AND DWIMFLG
                (ADDSPELL function.2))
           (RETURN function.2))
))
```

25.4.5 Undoing the Putting and Removing of Properties

PUTPROP (see Section 7.3) and REMPROP (see Section 7.4.2) are functions that put and remove properties from atoms. We would like to undo a PUTPROP if it overwites a property value that already exists. We would like to undo REMPROP if it removes a property from the atom's property list that should be retained. We may define these functions as follows:

```
(DEFINEQ
    (/putprop (atm property value)
```

```
(COND
     ((NULL atm)
          (ERROR "ATTEMPT TO RPLAC NIL"
                 (LIST atm property)))
     ((NOT (LITATOM atm))
          (ERROR "ARG NOT LITATOM"
                 atm)))
(PROG (prop.list.1 prop.list.2 temporary)
     (SETQ prop.list.1 (GETPROPLIST atm))
loop
     (COND
       ((NLISTP prop.list.1)
          (COND
            ((AND
                 (NULL prop.list.1)
                 prop.list.2)
             (SETQ temporary
                  (LIST property value))
             (AND
                  LISPXHIST
                  (UNDOSAVE
                       (LIST '/PUT-1
                             atm
                             temporary)
                       LISPXHIST))
             (/RPLACD (CDR prop.list.2)
                      temporary)
             (RETURN value))))
       ((NLISTP (CDR prop.list.1)))
       ((EQUAL (CAR prop.list.1) property)
          (/RPLACA (CDR prop.list.1) value)
          (RETURN value))
       (T
          (SETQ prop.list.2 prop.list.1)
          (SETQ prop.list.1 (CDDR
          prop.list.2))
          (GO loop)))
(SETQ temporary
     (CONS property
          (CONS value (GETPROPLIST atm))))
(AND LISPXHIST
     (UNDOSAVE
        (LIST '/put-1
              atm
              temporary)
```

```
                    LISPXHIST))
            (SETPROPLIST atm temporary)
            (RETURN value))
    ))
```

And the definition for /REMPROP might appear as:

```
(DEFINEQ
   (/remprop (atm property)
      (COND
          ((NULL (LITATOM atm))
              (ERROR "ATTEMPT TO RPLACA NIL" atm)))
      (PROG (prop.lst.1 prop.lst.2 value)
            (SETQ prop.lst.1 (GETPROPLIST atm))
       loop
            (COND
               ((OR (NLISTP prop.lst)
                    (NLISTP (CDR prop.lst)))
                        (RETURN value))
               ((EQUAL (CAR rop.lst) property)
                    (SETQ value property)
                    (AND LISPXHIST
                         (UNDOSAVE
                             (LIST '/PUT+1
                                   atm
                                   (CDR prop.lst.2)
                                   prop.lst.1)
                               LISPXHIST))
                    (COND
                       (prop.lst.2
                           (/RPLACD (CDR prop.lst.2)
                                    (CDDR prop.lst.1)))
                       (T
                           (SETPROPLIST atm
                                        (CDDR
                                        prop.lst.1))))
                    (SETQ prop.lst.1 (CDDR prop.lst.1)))
               (T
                    (SETQ prop.lst.2 prop.lst.1)
                    (SETQ prop.lst.1 (CDDR
                    prop.lst.2)))))
            (GO loop))
    ))
```

/PUT-1 may be defined as

```
(DEFINEQ
   (/put-1 (atm property)
     (PROG ((x atm))
      loop
           (COND
              ((EQUAL (CDR x) property)
                  (AND LISPXHIST
                       (UNDOSAVE
                          (LIST '/PUT-1
                                atm
                                x
                                property)
                          LISPXHIST))
                  (RPLACD x (CDDR property))
              ((SETQ x (CDR x))
                   (GO loop))))
   ))
```

25.4.6 Writing Your Own Undoable Functions

You may augment the functions that are undoable by the Programmer's Assistant by writing your undoable versions of your own functions. To do so, you will have to observe the following rules:

1. Use only undoable versions of the destructive functions in the functions that you write. If you need to know which functions are undoable, see the value of LISPXFNS in Section 25.8.
2. For your functions, write an undoable version for every function that modifies one of your data structures. The application user will use the regular versions, but LISPX will substitute the undoable versions for you.
3. Make the proper entries on LISPXFNS of the regular and undoable versions of your functions.

25.5 PROGRAMMER'S ASSISTANT FUNCTIONS

The Programmer's Assistant provides access to many functions that may be called from within your program. This allows you to create executive and input routines that are tailored to your applications. You may do this by redefining the basic INTERLISP functions, augmenting them with additional code, or calling them from your own executive routine.

25.5.1 LISPX Support Functions

LISPX invokes a number of support functions when it substitutes for INTERLISP's READ-EVAL-PRINT loop. These functions perform the same

functions as READ, EVAL, and PRINT, but are written to recognize the special commands supported by the Programmer's Assistant.

LISPXREAD is a substitute for the generalized READ function (see Section 14.1). It has the format

```
Function:       LISPXREAD

# Arguments:    2

Arguments:      1) a file name, FILE
                2) a read table, RDTBL

Value:          The next unread expression.
```

LISPXREAD obtains the next expression from the specified file. It may do this in one of two ways:

1. If READBUF is non-NIL, it contains a list of expressions that have been unread as the result of previous Programmer Assistant commands. LISPXREAD returns the next expression from this list.
2. If READBUF is NIL, LISPXREAD invokes READ (see Section 14.1) to acquire the next expression from the input stream.

LISPXREAD sets REREADFLG to NIL when it reads input via READ, but sets it to the value of READBUF when rereading.

The actions of LISPXREAD are mediated by a read table as described in Section 14.4.

A Definition for LISPXREAD

We might write LISPXREAD as follows:

```
(DEFINEQ
   (lispxread (file rdtbl)
      (PROG (lispxinput)
       loop
            (COND
               ((NULL READBUF)
                  (*
                        Nothing in the read buffer, so
                        to read from the terminal.
                  )
                  (SETQ REREADFLAG NIL)
                  (SETQ lispxinput
                        (COND
                           ((EQ lispxreadfn 'READ)
                                (READ T))
```

```
                    (T
                        (APPLY* lispxreadfn
                                file))))
            (COND
               ((READP T)
                  (*
                       Echo the input for the user
                       with no <CR> because there
                       is more input in the read
                       buffer.
                  )
                  (PRIN2 lispxinput T)
                  (SPACES 1 T))
               (T
                  (PRINT lispxinput T))))
            (RETURN lispxinput)))
      (SETQ lispxinput (CAR READBUF))
      (SETQ READBUF (CDR READBUF))
      (RETURN lispxinput))
))
```

However, there is a bit of subterfuge going on here. LISPXREAD actually does not do a READ if you are accepting data from a file. Rather, it performs the following statement:

```
(APPLY* LISPXREADFN file)
```

where LISPXREADFN is initially defined as READ. You may assign any reading function that you want to LISPXREADFN to perform input to your program. Since the reading function may parse the input stream, do character or macro substitution, or other operations, this provides you with a powerful mechanism for controlling how data is entered into your program from various sources. The reading function may even be configured to perform security and validation checks on the input file.

Unreading an Expression

LISPXUNREAD *unreads* a list of expressions contained in its argument, which is a list. It takes the form

```
Function:       LISPXUNREAD
# Arguments:    1
Argument:       1) a list of expressions to be unread, LST
Value:          The list of expressions.
```

LISPXUNREAD puts the unread expressions into READBUF. Thus, these expressions will be the next ones considered by LISPXREAD as input before going to the file. If the expressions which appear in this psuedo-input must be executed in a certain sequence, you are responsible for ordering them in the manner in which they are to be executed.

We might define LISPXUNREAD as follows:

```
(DEFINEQ
    (lispxunread (lst)
        (SETQ READBUF
            (COND
                ((NULL READBUF) lst)
                (T
                    (APPEND lst
                            (CONS HISTSTRO
                            READBUF)))))
    ))
```

Testing for Input

LISPXREADP is a predicate that tests whether anything is waiting to be read on the primary input stream. It takes the form

Function:	LISPXREADP
# Arguments:	1
Argument:	1) a flag, FLAG
Value:	T, if anything is waiting to be read; otherwise, NIL.

If FLAG has the value T, LISPXREADP will return T if there is any input waiting to be read via LISPXREAD. Thus, if the user types ahead to the read buffer, LISPXREAD will detect its presence.

If FLAG is NIL, LISPXREADP returns T if and only if there is input to be read on the current line. That is, it does not detect any expressions that you may have typed ahead.

In both cases, leading spaces are ignored by reading them with READC.

We might define LISPXREADP as follows:

```
(DEFINEQ
    (lispxreadp (flag)
        (COND
            (READBUF
                (OR flag
                    (NEQ (CAR READBUF) HISTSTRO)))
```

```
            (T
                (READP T)))
))
```

25.5.2 Evaluating Expressions as if LISPXREAD

When LISPXREAD reads input, the input is recorded on the history list as an event. You may enter events on the history list from your program by simulating the read and evaluation process. **LISPXEVAL** performs this simulation for you. It takes the form

Function:	LISPXEVAL
# Arguments:	2
Arguments:	1) a lispx expression, LISPXFORM 2) a lispx identifier, LISPXID
Value:	The value produced by evaluating LISPXFORM.

LISPXEVAL evaluates LISPXFORM as though it had been read by LISPX. That is, the expression is recorded as an event on the history list. The expression is made undoable by substituting the undoable versions of the corresponding destructive functions (see Section 25.4). The value produced by evaluating LISPXFORM is returned to the caller, but is not printed.

We might define LISPXEVAL as follows:

```
(DEFINEQ
    (lispxeval (lispxform lispxid)
        (PROG (lispxhist)
                (OR lispxid
                    (SETQ lispxid '←))
            (*
                Create the history list entry and
                make a copy to return to the caller.
            )
            (SETQ lispxhist
                (HISTORYSAVE LISPXHISTORY
                             lispxid
                             NIL
                             lispxform))
            (*
                Evaluate LISPXFORM and put value into
                history list entry.
            )
```

```
                (RPLACA lispxhist
                        (EVAL
                            (COND
                               ((NLISTP lispxform)
                                   lispxform)
                               (T
                                   (LISPX/ lispxform)))
                            lispxid))
                (*
                    Return the value computed from the
                    evaluation of LISPXFORM.
                )
                (RETURN (CAR lispxhist)))
    ))
```

25.5.3 Apprising the Assistant of Undoable Functions

When you have defined an undoable version of some destructive function, you need to apprise the Programmer's Assistant of its existence. You do this by executing **NEW/FN** with the name of the new function. It takes the form

```
Function:       NEW/FN
# Arguments:    1
Argument:       1) the name of an undoable function, FN
Value:          The name of the function.
```

NEW/FN updates the internal tables of the Programmer's Assistant so that a function will be undoable when typed in. The IRM suggests this example:

```
←(DEFINEQ
    (/radix (x)
        (UNDOSAVE (LIST '/radix (radix x)))
    ))
/radix
←(NEW/FN '/radix)
/RADIX
←(PRINT 150)
150
←(RADIX 12)
10
←(PRINT 150)
106
```

```
:                    (Note: the prompt is printed using
                     the radix also!)
(RESET)

←(PRINT 150)
150
```

25.5.4 Substituting Undoable Versions

LISPX automatically performs the substitution of undoable versions of destructive functions when it reads in an expression. To do so, it uses the function **LISPX/**, which takes the form

```
Function:       LISPX/

# Arguments:    3

Arguments:      1) an expression, EXPRESSION
                2) a function, FN
                3) a list of bound variables, VARS

Value:          The expression with appropriate
                substitutions.
```

If FN is non-NIL, it should be the name of a function. EXPRESSION will be its argument list.

If FN is NIL, then EXPRESSION is an expression in which substitutions are to be made.

In either case, VARS is an optional list of bound variables that may be used during the substitution process.

Consider the following examples:

```
←(LISPX/ '(SETQ X (CAR LETTERS)))
(SAVESET X (CAR LETTERS))

←(LISPX/ '(SETQ X (RPLACA HEAD (CAR TASK))))
(SAVESET X (/RPLACA HEAD (CAR TASK)))
```

25.5.5 Undoing Events

UNDOLISPX allows you to undo one or more events. It takes the form

```
Function:       UNDOLISPX

# Arguments:    1

Argument:       1) an event specification, EVENT

Value:          The name of the last function undone.
```

UNDOLISPX executes the UNDO command. EVENT is a list of one or more event specifications. UNDOLISPX invokes UNDOLISPX1 on each event in EVENT.

When you are undoing events, undoing them in order is guaranteed to restore you to the original state of your computation. However, undoing out of order is defined as restoring any cells changed in the indicated operation to their original state before the operation was performed. Undoing out of order will cause problems when several events are dependent on one another.

```
←(SETQ presidents '(hayes garfield polk buchanan))
(HAYES GARFIELD POLK BUCHANAN)

←(SETQ X (ATTACH presidents '(lincoln)))
((HAYES GARFIELD POLK BUCHANAN) LINCOLN)

←(SETQ Y (ATTACH X '(MCKINLEY)))
(((HAYES GARFIELD POLK BUCHANAN) LINCOLN) MCKINLEY)

←(UNDOLISPX '(2))
SETQ undone.

←X
UNBOUND ATOM
X

←(UNDOLISPX '(4))
UNDOLISPX undone.
UNDOLISPX

←X
((HAYES GARFIELD POLK BUCHANAN) LINCOLN)

←(UNDOLISPX '(6))
UNDOLISPX undone.
UNDOLISPX

←Z
((LINCOLN) MCKINLEY)
```

In general, operations will always be independent if they affect different lists or sublists (which are not tails) of the same list. Note that property lists are treated differently because each property is assumed to be independent.

```
←(SETQ REAGAN NIL)
NIL

←(PUTPROP 'REAGAN 'INAUGURAL-DATE 'JAN-20-1985)
JAN-20-1985

←(PUTPROP 'REAGAN 'TERM 'SECOND)
SECOND
```

```
←(PUTPROP 'REAGAN 'ELECTORAL-VOTES 525)
525

←GETPROPLIST 'REAGAN)
(INAUGURAL-DATE JAN-20-1985 TERM SECOND ELECTORAL-VOTES
525)

←(UNDOLISPX '(2))
PUTPROP undone.
PUTPROP

←(GETPROPLIST 'REAGAN)
(TERM SECOND ELECTORAL-VOTES 525)
```

Note that events are specified in a list. Here they have been specified by the number of the event, but you may also specify them by the name of the function used in the event.

Undoing Exactly One Event

UNDOLISPX1 allows you to undo exactly one event. It takes the form

```
Function:       UNDOLISPX1

# Arguments:    2

Arguments:      1) an event, EVENT
                2) a flag, FLAG

Value:          T or NIL.
```

UNDOLISPX1 undoes one event on a history list. It handles three cases:

1. If there is nothing to be undone, it returns NIL.
2. If the event was already undone, it prints the message "ALREADY UNDONE" and returns T.
3. Otherwise, it undoes the event, prints the appropriate message, and returns T.

If FLAG is T and the event is already undone, or is an UNDO command, UNDOLISPX1 does nothing but return NIL. When EVENT is NIL, UNDOLISPX searches the history list until it finds an event for which UNDOLISPX1 returns T.

To undo an event, UNDOLISPX1 must map down the property value for the SIDE field of the history list entries. For each element, it applies the CAR to the CDR and marks the event undone by attaching (with /ATTACH) a NIL to the front of the value of the SIDE property. Undoing elements in this fashion causes UNDOSAVE entries to be made on LISPXHIST. This allows you to undo the effects of UNDOLISPX1.

25.5.6 Undoing When Errors Occur

UNDONLSETQ is an NLAMBDA function that operates like NLSETQ. It takes the form

```
Function:       UNDONLSETQ
# Arguments:    1
Argument:       1) an expression, UNDOFORM
Value:          The value of UNDOFORM.
```

UNDONLSETQ evaluates UNDOFORM. If no error occurs, it returns (LIST (EVAL UNDOFORM)) and records the undo information on the history list. If an error does occur, UNDONLSETQ returns NIL (like NLSETQ) and undoes any changes that were made by undoable functions.

UNDO information is stored directly on the history event (if LISPXHIST is not NIL), so that if you execute a CTRL-D out of the UNDONLSETQ, the event will still be undoable.

The IRM suggests that UNDONLSETQ might be defined as follows:

```
(DEFINEQ
   (undonlsetq
      (NLAMBDA (undoform)
          (RESETLST
             (RESETSAVE (RESETUNDO)
                        (AND (EQUAL RESETSTATE 'ERROR)
                             (RESETUNDO OLDVALUE)))
             undoform)))
      ))
```

25.5.7 LISPX Printing Support

The Programmer's Assistant provides a number of printing functions to support display of information at the top-level executive. These functions have the same form

```
Function:       LISPXPRINT
                LISPXPRIN1
                LISPXPRIN2
                LISPXSPACES
                LISPXTERPRI
                LISPXTAB
                USERLISPXPRINT
# Arguments:    4
```

```
Arguments:      1) an expression, EXPRESSION
                2) a file name, FILE
                3) a read table, RDTBL
                4) a no print flag, NODOFLAG
Value:          The value of EXPRESSION.
```

Each of these functions performs the same action as its corollary as described in Chapter 15. They are redefined here to make entries on the history list and to provide a mechanism for modifying the top-level executive.

```
←(LISPXPRINT '(SETQ X (QUOTE (A B C D))) T T T)
NIL
```

because NODOFLAG is T.

Definitions for LISPX Printing Functions
We might define these functions as follows:

```
(DEFINEQ
   (lispxprint (expression file rdtbl nodoflag)
      (AND LISPXPRINTFLG
           LISPXHIST
           (LISPXPUT 'LISXPRINT
                     (LIST
                         (NLIST 'PRINT
                                expression
                                file
                                rdtbl))
                     T
                     LISPXHIST))
      (AND (NULL nodoflag)
           (PRINT expression file rdtbl))
   ))
(DEFINEQ
   (lispxprin1 (expression file rdtbl nodoflag)
      (AND LISPXPRINTFLG
           LISPXHIST
           (LISPXPUT 'LISXPRINT
                        (LIST
                        (NLIST 'PRIN1
                               expression
                               file
                               rdtbl))
```

```
                    T
                    LISPXHIST))
    (AND (NULL nodoflag)
         (PRIN1 expression file rdtbl))
))
```

and similarly for LIXPSPRIN2, LISPXTAB, LISPXSPACES, and LISPXTERPRI, which vary only in function name, argument to NLIST, and final print statement.

User-Defined LISPX Printing

USERLISPXPRINT permits you to define additional LISPX printing functions. It is defined to look back on the stack, find the name of the calling function, strip off the leading LISPX, perform the appropriate saving information, and then call the function to do the actual printing. The function called is the function name with the "LISPX" stripped off.

LISPXPRINTDEF

LISPXPRINTDEF is the corollary function to PRINTDEF. It takes the form

```
Function:       LISPXPRINTDEF

# Arguments:    6

Arguments:      1) an expression, EXPRESSION
                2) a file name, FILE
                3) a left-hand margin, LEFT
                4) a definition flag, DEFFLAG
                5) a tail flag, TAILFLAG
                6) a no print flag, NODOFLAG

Value:          The value of the expression.
```

All of the printing functions put their output on the history list under the property *LISPXPRINT*.

If NODOFLAG is non-NIL, these functions merely put their output on the history list, but do not print anything.

To perform output from a user program so that the output appears on the history list, you merely call the corresponding LISPX printing function rather than the standard printing function.

LISPX printing is mediated by the value of LISPXPRINTFLG. If NIL, the LISPX printing functions will not store their output on the history list.

25.6 CONTROLLING PROMPTING

History lists (see Chapter 28) contain numbered events that may be referred to by history commands. Normally, INTERLISP does not identify each statement

(also called "events" by the History Package). You may enable statement numbering by setting the variable PROMPT#FLG to T. The current event number will be printed preceding the prompt character. For example,

```
←MAKEFILE[AMISMAINT]
[DSK]AMISMAINT.;20

←(SETQ PROMPT#FLG T)
T

37←MAKEFILE[AMISUSER]
[DSK]AMISUSER.;7
```

Normally, a prompt is displayed by INTERLISP, you type an expression in response, and the expression is evaluated. That is, expressions are valuated after the prompt character has been displayed. You can force the evaluation of expressions before the prompt character is printed by assigning the expressions to be evaluated to PROMPTCHARFORMS. Initially, PROMPTCHARFORMS has the value (CHECKNIL). CHECKNIL is a system function that checks to see if the CAR or CDR of NIL have been reset or rebound. If so, it restores them and prints a warning message.

25.7 THE RESET PACKAGE

While your program is running, it may be interrupted or aborted because of errors or the pressing of control characters at the keyboard. The most devastating control character is CTRL-D because it forces a return to the top-level READ-EVAL-PRINT loop. However, your environment remains the same as established by your program. That is, any system variables that your program sets directly via SETQ or indirectly via a function call, and any global variables that your program defines will retain their values. Generally, you want to protect your environment by resetting variables to their original values when your program completes execution.

Errors may be caught (intercepted!) via ERRORSET. However, CTRL-Ds cannot be intercepted. The Reset Package provides functions to restore your environment in the event a CTRL-D or (RESET) is executed. However, you may also restore pieces of your environment when an error occurs using this package. While the Reset Package is useful when a (RESET) or CTRL-D occurs, it is quite valuable for performing selective restorations in the event of errors.

25.7.1 Establishing a Reset List

RESETLST establishes a reset list. It takes the following form

```
Function:       RESETLST
# Arguments:    1-N
```

```
Arguments:      1) a list of reset expressions, FORM[i]
Value:          Value of last form, FORM[n].
```

RESETLST is an NLAMBDA, nospread function. RESETLST sets up an ERRORSET (see Section 18.3) so that any reset operations performed by RESETSAVE will be restored when an error, (RESET), or a CTRL-D occurs.

If no error occurs, the value of RESETLST is the value of the last form. Otherwise, an error is generated after the necessary restorations are performed. That is, the environment before execution of the RESETLST will be restored if an error occurs.

25.7.2 Restoring Your Environment

RESETSAVE changes your environment by calling a function or setting a variable. At the same time, it sets up a specification for restoring the original value should an error or reset occur. RESETSAVE is usually used within a call to RESETLST. It takes the form

```
Function:       RESETSAVE
# Arguments:    2
Arguments:      1) an expression, X
                2) an expression, Y
Value:          Not a useful quantity (according to the
                IRM).
```

RESETSAVE is an NLAMBDA, nospread function, so its arguments are not evaluated before it is called. Rather, RESETSAVE evaluates its arguments according to the following cases:

X is Atomic
The top-level value of X is set to Y. Y is evaluated but X is not. When you enter the Editor, it performs

```
(RESETSAVE LISPXHISTORY EDITHISTORY)
```

to set the current history list to that used by the Editor. When you exit the Editor, the current history list will be restored as LISPXHIST.

X is Not Atomic
It is evaluated. If Y, e.g., the CDR of the form, is NIL, then X must return its former value while setting a new one. Thus, the effect of evaluating the form may be reversed when an error or reset occurs.

```
(RESETSAVE (CONTROL T))
```

will eliminate line buffering for the terminal table. It returns NIL, which is its normal state.

X is a Function
If X does not return its former value, the restoring expression is the value of Y. Y is evaluated before X.

```
(RESETSAVE (SETBRK ...)
           (LIST 'SETBRK (GETBRK)))
```

which restores the break characters by applying SETBRK to the value returned by (GETBRK). This expression is computed before (SETBRK) is evaluated.

X may be NIL
If Y is non-NIL. Y is executed whenever a reset occurs. The IRM notes that a useful form is

```
(RESETSAVE NIL (LIST 'CLOSEF ⟨filename⟩))
```

which closes a file whenever a reset occurs. Alternatively, you may wish to use

```
(RESETSAVE NIL (LIST 'CLOSEALL))
```

which closes all files whenever a (RESET) occurs.

MAKEFILEFORMS as an Example
In the File Package, the variable MAKEFILEFORMS is evaluated to set certain system variables whenever a MAKEFILE is performed. It inspects the options passed to MAKEFILE to set these variables. To ensure that their values are reset when MAKEFILE exits, they are set inside a RESETSAVE as follows:

```
(ADDVARS
    (MAKEFILEFORMS
        (COND
            ((MEMBER 'NOCLISP OPTIONS)
                (RESETSAVE PRETTYTRANFLG T))
            ((MEMBER 'CLISP% OPTIONS)
                (RESETSAVE PRETTYTRANFLG 'BOTH)))
        (COND
            ((MEMBER 'FAST OPTIONS)
                (RESETSAVE PRETTYFLG NIL)))
        (COND
            ((OR
```

```
                    (MEMBER 'CLISPIFY OPTIONS)
                    (MEMBER 'CLISP OPTIONS))
                    (RESETSAVE CLISPFIYPRETTYFLG T))
               ((OR
                    (EQUAL FILETYPE 'CLISP)
                    (MEMBER 'CLISP (LISTP FILETYPE)))
                    (RESETSAVE CLISPIFYPRETTYFLG
                    'CHANGES)))
          (AND
               (NEQ (LINELENGTH) FILELINELENGTH)
               (RESETSAVE LINELENGTH FILELINELENGTH))))
```

25.7.3 Resetting Variables

RESETVAR allows you to save and restore global variables. It actually combines the effects of RESETLST and RESETSAVE in a simplified form. It takes the form

```
Function:       RESETVAR
# Arguments:    3
Arguments:      1) a variable, VAR
                2) a new value, VALUE
                3) an expression, EXPR
Value:          The value of EXPR.
```

RESETVAR is an NLAMBDA function. The function call

```
(RESETVAR var value expr)
```

is equivalent to executing

```
(RESETLST (RESETSAVE var value) expr)
```

You may evaluate a function with regard to EDITHISTORY rather than LISPXHISTORY using the following form:

```
(RESETVAR LISPXHISTORY EDITHISTORY (⟨function⟩))
```

Consider the following example from FNVARS which defines the INTERLISP environment at loadup:

```
(??T
   (PROG (TEM)
     (RESETVAR
```

```
        PRETTYTRANFLG
        T
        (PRINTDEF
           (COND
              ((NULL (CDAR (SETQ TEM
                                 (LISPXFIND LISPHISTORY
                                            LISPXLINE
                                            'ENTRY
                                            T))))
                     (CAAR TEM))
              (T (CAR TEM)))
           NIL
           T))
    (TERPRI T)
    (RETURN (CADDR TEM))))
```

which implements the ??T macro of the History Package.

An Alternative Form for RESETVARS

An alternative form, **RESETVARS**, operates like a PROG form on global variables. It takes the form

```
Function:       RESETVARS
# Arguments:    1-N
Arguments:      1) a global variable list, VARSLST
                2) a set of expressions, EXPRESSION[i]
Value:          The value of EXPRESSION[n].
```

RESETVARS sets up a RESETSAVE for each of the variables mentioned in VARSLST. Note that this function is treated differently in shallow-bound (INTERLISP-10/VAX/370) versus deep-bound (INTERLISP-D) systems; you should consult the IRM for more information.

The purpose of RESETVARS is to allow transportability of programs from shallow-bound to deep-bound systems, and vice versa.

25.7.4 Resetting Expressions

RESETFORM is a shorthand for RESETLST and RESETSAVE when the function returns its previous setting. It takes the form

```
Function:       RESETFORM
# Arguments:    2-N
```

```
Arguments:      1) a list of expressions, RESETFORMX
                2) a set of expressions, EXPRESSION[i]

Value:          If no error occurs, the value returned by
                evaluating EXPR[n].
```

RESETFORM is an NLAMBDA, nospread function. It is used when the corresponding function returns as its value the previous setting of some internal variable. It is equivalent to executing

```
(RESETLST
     (RESETSAVE ⟨resetformx⟩
     expression[1]
     ...
     expression[N])
```

Consider the following example from FNVAR:

```
(GREETFORM
   '(LISPXEVAL
      (PROGN
         (SETQ RESETFORMS
               (REMOVE GREETFORM RESETFORMS))
         (COND
            ((NEQ SYSTEMTYPE
                  (SETQ SYSTEMTYPE (SYSTEMTYPE)))
                  (SELECTQ
                     (TOPS20
                        (SETQQ SYSOUT.EXT EXE))
                     (TENEX
                        (SETQQ SYSOUT.EXT SAV))
                     (SHOULDNT))
                  (RESTETERMCHARS)
                  (RESETTERMCHARS ASKUSERTTBL)))
         'GREET))
      '←)
```

25.7.5 Establishing UNDO Information

RESETUNDO initializes the saving of UNDO information when used in a resetlist. It takes the form

```
Function:       RESETUNDO

# Arguments:    2
```

```
Arguments:      1) an expression, EXPRESSION
                2) a stop flag, STOPFLAG
Value:          A value for undoing side effects.
```

The value produced by RESETUNDO, when given back to RESETUNDO, will cause any intervening side effects to be undone.

```
(RESETLST
     (RESETSAVE (RESETUNDO)) . ⟨forms⟩)
```

will undo the side effects of evaluating ⟨forms⟩ when a normal exit occurs, when an error occurs, or a CTRL-D is executed.

If STOPFLAG has the value T, RESETUNDO stops accumulating undo information that it has been saving about the expression.

(RESETUNDO) initializes the saving of undo information.

25.7.6 Structure of RESETFORMS

RESETFORMS is a list of expressions that is evaluated whenever a system reset occurs, either via (RESET) or CTRL-D. The structure of RESETFORMS is

```
((SETQ READBUF NIL)
 (SETQ READBUFSOURCE NIL)
 (SETQ TOPLISPXBUFS
       (OR (CLBUFS T)
           TOPLISPXBUFS))
 (COND
   ((EQ CLEARSTKLST T)
      (COND
        ((EQ NOCLEARSTKLST NIL)
             (CLEARSTK))
        (T
             (MAPC (CLEARSTK T)
                   (FUNCTION
                      (LAMBDA (X)
                         (AND
                              (NOT (MEMB X
                              NOCLEARSTKLST))
                              (RELSTK X)))))))))
   (T
      (MAPC CLEARSTKLST (FUNCTIONRELSTK))
      (SETQ CLEARSTKLST NIL)))
 (SETQ STACKOVERFLOW))
```

25.8 PROGRAMMER'S ASSISTANT VARIABLES

The Programmer's Assistant uses a number of variables to control the evaluation of input.

LISPXFNS — This variable is a list of all the functions for which LISPX will substitute a function if the function name read appears in the list. Principally, it is used to determine when to substitute undoable versions of functions for basic INTERLISP functions.

```
←LISPXFNS
((SETQ . SAVESETQ)  (SET . SAVESET) (SETQQ . SAVESETQQ)
(DEFINEQ . /DEFINEQ) (DEFINE . /DEFINE) (PUT . SAVEPUT)
(PUTPROP . /PUTPROP) (RETFROM . BREAKRETFROM) (RETEVAL .
BREAKRETEVAL) (ADDPROP . /ADDPROP) (ATTACH . /ATTACH)
(CLOSER . /CLOSER) (CONTROL . /CONTROL) (DELETECONTROL .
/DELETECONTROL) (DREMOVE . /DREMOVE) (DREVERSE .
/DREVERSE) (DSUBST . /DSUBST) (ECHOCONTROL . /ECHOCONTROL)
(ECHOMODE . /ECHOMODE) (FNCLOSER . /FNCLOSER) (FNCLOSERA
. /FNCLOSERA) (FNCLOSERD . /FNCLOSERD) (LCONC . /LCONC)
(LISTPUT . /LISTPUT) (LISTPUT1 . /LISTPUT1) (MAPCON .
/MAPCON) (MAPCONC . /MAPCONC) (MOVD . /MOVD) (NCONC .
/NCONC) (NCONC1 . /NCONC1) (PUTASSOC . /PUTASSOC) (PUTD .
/PUTD) (PUTDQ . /PUTDQ) (PUTHASH . /PUTHASH) (RADIX .
/RADIX) (RAISE . /RAISE) (REMPROP . /REMPROP) (RPLACA .
/RPLACA) (RPLACD . /RPLACD) (RPLNODE . /RPLNODE) (RPLNODE2
. /RPLNODE2) (SETA . /SETA) (SETATOMVAL . /SETATOMVAL)
(SETBRK . /SETBRK) (SETD . /SETD) (SETPROPLIST .
/SETPROPLIST) (SETREADTABLE . /SETREADTABLE) (SETSEPR .
/SETSEPR) (SET SYNTAX . /SETSYNTAX) (SETTERMTABLE .
/SETTERMTABLE) (SETTOPVAL . /SETTOPVAL) (TCONC . /TCONC)
(REPLACEFIELD . /REPLACEFIELD) (DELFILE . /DELFILE)
(UNDELFILE . /UNDELFILE))
```

LISPXCOMS — This variable is used to correct the spelling of Programmer's Assistant commands.

```
←LISPXCOMS
(XPR XPRESS DIR $ ... ?? FIX FORGET NAME ORIGINAL REDO
REPEAT RETRY UNDO USE fix forget name redo repeat retry
undo use SHH RETRIEVE BEFORE AFTER TYPE-AHEAD ??T CONTIN)
```

LISPXPRINTFLG	This variable determines whether the LISPX printing functions save their output on the history list or not. If it is NIL, the output will not be saved.
LISPXHISTORY	The standard history list for top level input to INTERLISP.

25.9 LISPX STATISTICS

In INTERLISP-10, the Programmer's Assistant keeps various statistics about system usage.

25.9.1 Printing LISPX Statistics

You may print a summary of the LISPX statistics using **LISPXSTATS**, which takes the form

Function:	LISPXSTATS
# Arguments:	1
Argument:	1) a value return flag, RETURNVALUESFLAG
Value:	A list of statistics or NIL.

LISPXSTATs will print a summary of the statistics accumulated by the Programmer's Assistant at your terminal. If RETURNVALUESFLAG is T, the statistics will be returned as a list where each element has the form

(⟨value⟩ . ⟨explanation⟩)

Consider the following example:

```
←(LISPXSTATS)
(32 LISPX INPUTS)
```

```
(153 UNDO SAVES)
(20 CHANGES UNDONE)

(1 EDIT UNDO SAVES)

(5 CALLS TO DWIM)
(5 WERE DUE TO ERRORS)

OF THOSE DUE TO ERRORS:
(5 WERE DUE TO ERRORS IN TYPE-IN)

OF THE CALLS DUE TO DWIMIFYING:

(0:21:3 CONSOLE TIME)
(0:0:0 OF IT IN THE EDITOR)
(0:0:5 CPU TIME)
(0:0:) OF IT IN DWIM)
T
```

25.9.2 Adding New Statistics

You may add new statistics to those kept by the Programmer's Assistant using **ADDSTATS**, which takes the form

```
Function:       ADDSTATS

# Arguments:    1-N

Arguments:      1-N) a statistic entry, STATISTIC[i]

Value:          The value of STATISTIC.
```

ADDSTATS is an NLAMBDA, nospread function. Each STATISTIC[i] is a list of the form

```
(⟨statistic-name⟩ . ⟨message⟩)
```

where each ⟨statistic-name⟩ defines a new statistic.

```
←(ADDSTATS (EDITOR-CALLS CALLS TO THE EDITOR))
((EDITOR-CALLS CALLS TO THE EDITOR))
```

25.9.3 Updating Statistics

Both the user and the Programmer's Assistant may update statistics variables using **LISPXWATCH**, which takes the form

```
Function:       LISPXWATCH

# Arguments:    2
```

```
Arguments:      1) a statistic name, STATISTIC
                2) a number, N

Value:          The new value of the statistic.
```

LISPXWATCH updates the specified statistic by N. If N is NIL, a default of 1 is assumed. Consider the following example:

```
←(LISPXWATCH 'EDITOR-CALLS 2)
NIL

←(LISPXSTATS)
...
(2 CALLS TO THE EDITOR)
T
```

because new statistics are added to the end of the list SYSTATS which contains the format for the LISPX statistics.

25.9.4 System Statistics

The Programmer's Assistant currently supports the following statistics which are kept as elements of the list SYSTATS:

```
←SYSTATS
((LISPXSTATS          LISPX INPUTS)
 (UNDOSAVES           UNDO SAVES)
 (UNDOSTATS           CHANGES UNDONE)
 (EDITCALLS           CALL TO EDITOR)
 (EDITSTATS           EDIT COMMANDS)
 (EDITUNDOSAVES       EDIT UNDO SAVES)
 (EDITUNDOSTATS       EDIT CHANGE SUNDONE)
 (P.A.STATS           P.A. COMMANDS)
 (CLISPIFYSTATS       CALLS TO CLISPIFY)
 (FIXCALLS            CALLS TO DWIM)
 (ERRORCALLS          WERE DUE TO ERRORS)
 (DWIMIFYFIXES        WERE FROM DWIMIFYING)
 (FIXTIME)
 NIL
 "OF THOSE DUE TO ERRORS:"
 (TYPEINFIXES         WERE DUE TO ERRORS in TYPE-IN)
 (PROGFIXES           WERE DUE TO ERRORS IN USER PROGRAMS)
 (SUCCFIXES1          OF THESE CALLS WERE SUCCESSFUL)
 NIL
 "OF THESE CALLS DUE TO DWIMIFYING:"
 (SUCCFIXES2          WERE SUCCESSFUL)
```

```
(SPELLSTATS         OF ALL DWIM CORRECTIONS WERE SPELLING
                    CORRECTIONS)
(CLISPSTATS         WERE CLISP TRANSLATIONS)
(INFIXSTATS         OF THESE WERE INFIX TRANSFORMATIONS)
(IFSTATS            WERE IF/THEN/ELSE STATEMENTS)
(I.S.STATS          WERE ITERATIVE STATEMENTS)
(MATCHSTATS         WERE PATTERN MATCHES)
(RECORDSTATS        WERE RECORD OPERATIONS)
(SPELLSTATS1        OTHER SPELLING CORRECTIONS, E.G.,
                    EDIT COMMANDS)
(RONONSTATS         OF ALL SPELLING CORRECTIONS WERE RUN-
                    ON CORRECTIONS)
(VETOSTATS          CORRECTIONS WERE VETOED))
```

26

Masterscope

One of the most useful techniques for programmers is the ability to analyze source programs. Conventional languages often provide such support for source programs. Implementation is difficult because of the wide variety of data structures and statements one encounters in a traditional programming language such as FORTRAN or PASCAL. This diversity means that source program analysis systems become quite complex because they must deal with different types of information for each statement. Few conventional programming languages support this facility at run-time because of the extensive amount of information required to answer ad hoc queries.

INTERLISP, because it is an integrated programming environment, removes the dichotomy between source and run-time programs. This allows it to interface with the Editor and the File Package to dynamically track changes to programs as they are created and modified.

Masterscope maintains a database of the relations that it "notices" when a function is analyzed. It develops sets of primitive relations, such as USE AS PROPERTY, which it modifies as changes are made to the source code. The resulting database allows you to determine

What functions a given function calls

How and where variables are bound

How and where variables are set and/or referenced

Which record declarations are used by functions.

You may interrogate the Masterscope database via a simple command language. You may also display the hierarchy of invocation for any function.

Because Masterscope analyzes only source programs, it will not work on compile code. That is, only EXPR definitions of functions are analyzed. If there is no in-core definition, Masterscope attempts to read the definition from a file that it knows about (see Chapter 17). It will search only those files that have been noticed or are represented on FILELST.

Masterscope is interfaced with both the Editor and the File Package so that when a function is edited or a new definition(s) loaded into your environment, Masterscope knows that it must (re)analyze the function(s).

26.1 MASTERSCOPE CONCEPTS

Masterscope relies on a few basic concepts to support the analysis of programs. These are

Relations
A relationship between functions and variables establishing usage and/or interaction.

Set
A collection of "things" to be operated on by a Masterscope command (see below).

Path
The hierarchical control structure through a program whence execution may flow.

Template
Templates are patterns that describe how a function is evaluated.

26.2 INTERACTING WITH MASTERSCOPE

You may interact with Masterscope via an English-like command language. Masterscope commands consist of verbs that direct it to answer questions using the database or to perform analyses that result in creating or updating the database. This section defines the various commands. Section 26.3 discusses set specifications.

26.2.1 Analyzing Functions

ANALYZE directs Masterscope to analyze the functions given as its arguments. It takes the form

```
←. ANALYZE ⟨set specification⟩
←. REANALYZE ⟨set specification⟩
```

The set specification identifies the functions to be analyzed. The information is entered into the database. Functions called by functions (ad infinitum) described by the set specification are included in the analysis. Masterscope does not analyze functions that it thinks it already knows about (e.g., that have infor-

mation already in the database). To force it to look at a function again, you must execute the REANALYZE command.

Consider the following example:

```
←. ANALYZE INHERIT
.....................done
←. SHOW PATHS FROM INHERIT
1. INHERIT ALLSLOTS?
2.         ALLDEMONS?
3.         INHERIT-SLOT SLOT? ALLSLOTS?
4.       |   PUTVALUE PASSITDOWN DO-OFFSPRING DO-SUBNODE [a]
5.       |     |         ALLSLOTS?
6.       |     MAKESLOT ENTER
7.       |     |         MS1 MS2 MAKESLOT [6]
8.       |     GETVALUE
9.       |     REPLACADDD REPLACA
10.      INHERIT-DEMON SLOT? [3]
11.          MAKEDEMON PASSDEMONDOWN DO-OFF-OF-DEMON [b]
12.          REPLACADDD [9]
------------------------------------------ overflow-a
13. DO-SUBNODE SLOT? [3]
14.            MAKESLOT [6]
15.            REPLACADDD [9]
16.            ADDVALUE PASSITDOWN [4]
17.                     ALLSLOTS?
18.                     GETVALUE
------------------------------------------ overflow-b
19. DO-DEMON MAKEDEMON [11]
20.          REPLACADDD [9]
```

Whenever a function is referred to in a command as a subject of a relation, it is automatically analyzed if an analysis has not previously been performed.

If Masterscope cannot find the function, it displays the message "⟨function⟩ CAN'T BE ANALYZED". If the function was previously known (described in the database), but cannot now be found (because its definition has been reset or the file closed) Masterscope displays the message "⟨function⟩ DISAPPEARED!".

You may also analyze the functions in a file. A number of forms are available to perform this operation. Consider

```
←. ANALYZE ALL IN AMISNETFNS
```

which analyzes all of the functions contained in the list that is the value of AMISNETFNS. If the file contains individual (FNS ...) commands as part of its File Package commands, you may use the form

```
←. ANALYZE ALL IN (FILEFNSLST 'AMISDB)
```

where FILEFNSLST (see Section 17.7.3) gathers the individual FNS commands together to form one list.

26.2.2 Erasing the Database

You may delete information from the database by executing the **ERASE** command, which takes the form

```
←. ERASE ⟨set specification⟩
```

ERASE removes all information described by the set specification from the database:

```
←. ERASE INHERIT
OK
←. WHO CALLS INHERIT
NIL
```

Caution: Executing ERASE without any arguments is tantamount to clearing the entire database. This occurs because the set specification is null which implies that everything should be removed.

26.2.3 Showing Structures

Masterscope allows you to display the structure of a program (or a subset of it) either as a tree or a list of functions and/or variables.

The **SHOW PATHS** command displays the tree of functions (as above) using a path specification. It takes the form

```
←. SHOW PATHS ⟨path specification⟩
```

Path specifications are described in Section 26.5. The example in Section 26.2 shows a simple execution of SHOW PATHS for the function INHERIT.

The **SHOW WHERE** command allows you to display those functions that meet specified criteria given by a structure specification. The format of the SHOW WHERE command is

```
←. SHOW WHERE ⟨set1⟩ ⟨relation⟩ ⟨set2⟩
```

where the form ⟨set1⟩⟨relation⟩⟨set2⟩ is the structure specification. ⟨set1⟩ must refer to a set of functions. ⟨set2⟩ may refer to functions or variables.

Sets and relations are described in Sections 26.3 and 26.4 respectively.

```
←. ANALYZE ALL IN COMPLEXFNS
.....................done
←. SHOW WHERE REAL IS CALLED
CPLUS:
     (REAL CX1)
     (REAL CX2)
CDIFFERENCE:
     (REAL CX1)
     (REAL CX2)
CMULT:
     (REAL CX1)
     (REAL CX2)
     (REAL CX1)
     (REAL CX2)
PRINT.COMPLEX:
     (REAL CX1)
done
```

The **DESCRIBE** command allows you to display information about functions described in a set specification. It takes the form

```
←. DESCRIBE ⟨set specification⟩
```

DESCRIBE prints out the bindings of variables in the function, the variables used freely by the function, and the functions called by the function. Consider the following example:

```
←.DESCRIBE INHERIT

INHERIT[NODE,FROM,ITYPE]
  calls:        MEMB,PRINT,MAPCAR,PUTPROP,ALLSLOTS?,
                ALLDEMONS?, ASSOC,GETPROP,CONS,LIST,SUBST,
                INHERIT-SLOT, INHERIT-DEMON
  binds:        OFFLIST,IPROPS
  uses free:    ALLNODES
  file:         NETWORK.LISP
```

26.2.4 Editing Functions

Masterscope allows you to perform global editing of functions using set or structure specifications. The two forms of the EDIT command are

```
←. EDIT WHERE ⟨set1⟩⟨relation⟩⟨set2⟩
              [-⟨editcommands⟩]
```

and

```
←. EDIT ⟨set⟩ [-⟨editcommands⟩]
```

Set and relation specifications are discussed in Sections 26.3 and 26.4, respectively.

⟨editcommands⟩ is an optional list of edit commands that are applied to each function in turn.

26.2.5 Checking Sets

You may check a file for various anomalous conditions. CHECK takes the form

```
←. CHECK ⟨set⟩
```

If ⟨set⟩ is not given, FILELST is used. Consider the following example:

```
←. CHECK AMISNET

⟨⟨⟨⟨,
⟨ In (no block-DEMON? KILLDEMON MAKEDEMON ...) ⟩⟩⟩⟩⟩
(note)
NODE-not declared, used freely by-INHERIT.SLOT
INHERIT.DEMON DO.SUBNODE, etc.
INHERITANCE-" "-INHERIT.SLOT INHERIT.DEMON
GIVE.TO.OFFSPRING etc.
(possible error)
← ⟨ ⟩ -
not declared, never bound, no top-level value, used
freely by SHOW.OPEN.DATABASE.STRUCTURE
!  -not bound, not a GLOBALVAR, used freely by-
SHOW.OPEN.DATABASE.STRUCTURE
```

Note that the last two errors result from CLISP structures which are not properly recognized by Masterscope.

26.2.6 Using CLISP in Masterscope

You may use the CLISP iterative statement operators in Masterscope through the FOR command. It takes the form

```
←. FOR ⟨variable⟩ ⟨set⟩ ⟨i.s.tail⟩
```

An iterative statement is constructed where ⟨variable⟩ is iteratively assigned to each element of ⟨set⟩ and then the iterative statement tail ⟨i.s.tail⟩ is executed.

```
←. FOR X CALLED BY MY.EXECUTIVE WHEN CCODEP
       DO (PRINTOUT T X ... (ARGLIST X) T)
```

which prints out the name and arguments of all compiled functions that are called by the function MY.EXECUTIVE.

26.2.7 Obtaining Help

You may obtain rudimentary help by executing the **HELP** command, which takes the form

```
←. HELP
```

This command prints out a summary of the commands available in Masterscope.

```
←.HELP
*-------------------------------------------------------*
a ⟨command⟩ is:

[RE]ANALYZE ⟨functions⟩
ERASE ⟨functions⟩
SHOW PATHS ⟨pathoptions⟩
⟨set⟩ [⟨relation⟩ | IS | ARE] ⟨set⟩
EDIT WHERE ⟨functions⟩ [⟨relations⟩ ⟨set⟩] [-⟨edit
commands⟩]
SHOW WHERE ⟨functions⟩ ⟨relation⟩ ⟨set⟩
CHECK ⟨files⟩
FOR ⟨variable⟩ ⟨set⟩ ⟨iterative statement tail⟩

-------------------------------------------------------
a ⟨set⟩ is (at least one of):
a determiner +      a type          +    a specification

THE                 FUNCTIONS             ['] [⟨atom⟩ | ⟨list⟩]
ANY                 VARIABLES             @ ⟨predicate⟩
WHICH               PROPERTY NAMES        IN ⟨expression⟩
WHO                 RECORDS               ⟨relation⟩ING ⟨set⟩
                    FIELDS                ⟨relation⟩ED [BY |
                                          IN] ⟨set⟩
                    FILES                 THAT ⟨relation⟩
                                          ⟨set⟩
```

```
                    I.S.OPRS            LIKE ⟨edit pattern⟩
                                        ON ⟨files⟩
                                        ON PATH
                                        ⟨pathoptions⟩
                    FIELDS OF ⟨records⟩
                    ⟨blockword⟩ [ON ⟨files⟩ | OF
                    ⟨functions⟩]
⟨functions⟩, ⟨files⟩, etc. are ⟨set⟩s whose type is
implied.
-------------------------------------------------------
a ⟨relation⟩ is a verb and optional modifier:

verbs:      modifiers (anywhere after the verb):

CALL        [SOMEHOW | FOR EFFECT | FOR VALUE
             DIRECTLY | INDIRECTLY]
USE         AS [RECORD | PROPERTY | ⟨record⟩ FIELD]
            ⟨name⟩
USE         AS CLISP ⟨word⟩
USE         [FREELY | LOCALLY]
SET         [FREELY | LOCALLY]
SMASH       [FREELY | LOCALLY]
TEST        [FREELY | LOCALLY]
REFERENCE   [FREELY | LOCALLY]
DECLARE     AS [LOCALVAR | SPECVAR]
BIND
FETCH
REPLACE
CREATE
CONTAIN
-------------------------------------------------------
⟨pathoptions⟩              |    abbreviations & synonyms
FROM ⟨function⟩            |    FNS = FUNCTIONS
TO ⟨functions⟩             |    PROPS = PROPERTIES
AVOIDING ⟨functions⟩       |    VARS = VARIABLES
NOTRACE ⟨functions⟩        |    (& singular FN, VARIABLE,
                           |    etc)
SEPARATE ⟨functions⟩       |    FREE = FREELY
LINELENGTH ⟨number⟩        |    LOCAL = LOCALLY
                   |    AMONG = AVOIDING NOT
-------------------------------------------------------
⟨sets⟩ may be joined by AND or OR or preceded by NOT.
Any commands can be followed by OUTPUT ⟨filename⟩.
*-----------------------------------------------------*
```

26.3 SPECIFYING SETS

A *set specification* is a phrase that describes a set to be operated upon by Masterscope commands. A set specification consists of a *determiner*, a *type*, and a *specification*.

A *determiner* is one of following: THE, ANY, WHICH, WHO, or WHOM. It defines the scope of the set. If the determiner is omitted, ANY is assumed. Set phrases may be preceded by a determiner. WHICH and WHO are interrogative determiners that are only valid for certain commands. ANY, WHO, and WHOM can be used alone as they are wild-card elements.

A *type* is an INTERLISP datatype. The datatypes currently supported are FUNCTIONS, VARIABLES, PROPERTY NAMES, RECORDS, FIELDS, or FILES. Any set phrase has a type, e.g., the INTERLISP datatype of the objects that the command will operate upon. Types are used as follows:

1. Set types are used to differentiate between possible parsings. Consider the examples

   ```
   WHO SETS ANY BOUND IN INHERIT OR USED BY GETVALUE

   WHO SETS ANY BOUND IN INHERIT OR CALLED BY GETVALUE
   ```

   Both of these commands have the same form. However, the first example is parsed as

   ```
   WHO SETS ANY (BOUND BY INHERIT OR USED BY GETVALUE)
   ```

   since BOUND BY and USED BY refer to variables, while the second example is parsed as

   ```
   WHO SETS ANY BOUND IN (INHERIT OR CALLED BY GETVALUE)
   ```

   since CALLED BY and INHERIT must refer to functions.
2. The type is used to determine the modifier for USE.
3. The interpretation of CONTAIN(S) depends on the type of the object. For example, WHAT FUNCTIONS ARE CONTAINED IN FRAMEFNS. WHO IS CONTAINED IN FRAME would print all object contained in the file FRAME.
4. The context in which a set expression is interpreted depends on the type. For example, ANY VARIABLES @ GETD is interpreted as the set of all variables that have been noticed by Masterscope as being bound or utilized in any function which has been analyzed.

Sets may be joined by AND or OR or preceded by NOT to form new sets. AND is always interpreted as meaning the intersection of the two sets, i.e., all elements must satisfy both set specifications. OR is interpreted as the union of the two sets, i.e., any element must satisfy either of the two set specifications. NOT is interpreted as the complement of the set specification.

```
WHO CALLS GETVALUE AND PUTVALUE
```

means all those functions which call both GETVALUE and PUTVALUE.

You may not join modifiers with conjunctions in Masterscope even though these constructs are allowed in English.

A *specification* is a description of the characteristics of the set. Specifications are drawn from the following list:

'⟨atom⟩	You may specify a single object by its name. For example, WHO CALLS 'MAKE-SLOT will display the functions that invoke the function MAKE-SLOT. You may leave the ' out of the specification, but this form may yield ambiguities. Good practice suggest that you always quote the atom.
'⟨list⟩	You may specify several atoms by including them in a list structure. For example, WHICH FUNCTIONS CALL '(GETVALUE PUTVALUE) which returns a list of the functions that call either GETVALUE or PUTVALUE or both.
IN ⟨expression⟩	You may specify an S-expression to be evaluated which returns a list of elements specifying the set. For example, IN ASPTPFNS

returns a list of functions associated with the file ASPTP via the File Package specification.

@⟨predicate⟩ — You may specify a predicate which the elements of the set must satisfy. The ⟨predicate⟩ may be a function name, either system or your own definition, a LAMBDA expression, or an expression in terms of the variable X. For example,

```
WHO IN FRAMEFNS CALLS ANY NOT
@GETD
```

should display all functions having EXPR definitions.

The @ in the specification indicates that members of the set are those functions for which the predicate is non-NIL.

LIKE ⟨atom⟩ — The ⟨atom⟩ is used as a pattern to be matched against the names of function. For example,

```
WHO LIKE /R$ IS CALLED BY ANY
```

yields the functions /RPLACA and /RPLNODE.

⟨relation⟩ING ⟨set⟩ — ⟨relation⟩ is one of USE, CALL, BIND, SET possibly modified (SEE Section 26.4). The specified set consists of those objects that have the specified relation to ⟨set⟩. For example,

```
USING 'ALLNODES FREELY
```

would yield INHERIT.

⟨relation⟩ED BY ⟨set⟩ — Similar to ⟨relation⟩ING except

⟨relation⟩ED IN ⟨set⟩	that it finds those objects operated upon rather than on. For example, USED FREELY BY ANY CALLING GETVALUE
FIELDS OF ⟨set⟩	⟨set⟩ is a collection of record names. The fields of each record are to be operated on. WHO USES FIELDS OF SLOTRECORD
KNOWN	This verb implies the set of all functions that have been analyzed. For example, WHO IS KNOWN prints out a list of all of the functions that have been analyzed.
THOSE	This command implies the set created by the last Masterscope command. For example, WHO IS USED FREELY BY ASSERTION-RESOLVE WHO BINDS THOSE would find where the specified variables are bound.
ON PATH ⟨pathoptions⟩	This option implies a set constructed by the specified ⟨pathoptions⟩ (see Chapter 26.5). For example, IS NODE BOUND BY ANY ON PATH TO 'GETVALUE The value of this specification is the same as that created by SHOW PATHS command with similar arguments.

26.4 SPECIFYING RELATIONS

Relations are comparisons between sets. A relation helps to constrain or identify the set of objects to be considered for further analysis. A relation is implemented as a verb. Some verbs may accept modifiers. All may be used in the present tense or as present or past participles.

CALL

A function ⟨fn1⟩ calls a function ⟨fn2⟩ if the definition of ⟨fn1⟩ contains one or more of the following expressions:

(⟨fn2⟩ ⟨arguments⟩)
(APPLY ⟨fn2⟩ ⟨arguments⟩)
(FUNCTION ⟨fn2⟩)

```
←. WHO CALLS WHOM IN (FILEFNSLST 'AMISNET)
EDIT.SLOT.OR.DEMON--(IS.SLOT.NAME? IS.DEMON.NAME?)
EXECUTE.UPDATE.FUNCTION--(IS.MODEL.NAME? IS.SLOT.NAME?
IS.DEMON.NAME?)
UNLOAD.AND.CLOSE.DATABASE--(CHECK.OPEN.DATABASE
PRINT.DB.MESSAGE)
MAKENODE--(ADD.TO.DB.VARS INITIALIZE.NODE)
...
etc.
```

CALL SOMEHOW

A function ⟨fn1⟩ calls another function ⟨fn2⟩ somehow if there is some path from the first function to the other. Several levels of indirection may be necessary to determine if ⟨fn1⟩ calls ⟨fn2⟩. Masterscope recomputes this relationship dynamically because of the amount of information that must be retained in order to explore the possible paths to other functions.

SET

A function sets a variable if the function contains one or more expressions of the form

(SETQ ⟨variable⟩ ⟨expression⟩)
(SETQQ ⟨variable⟩ ⟨expression⟩)
(i.e, see Sections 3.8 and 3.9).

```
←. WHO SETS WHOM
SHOW.DATA.MODEL.OPERATIONS -- (A.DB.NODE)
INITIALIZE.NEW.DATABASE -- (CURRENT.DATABASE.NAME
CURRENT.DB.VARS CURRENT.DB.PROPS CURRENT.DB.FNS)
done
```

SMASH

A function smashes a variable if the function performs a destructive replacement of the value of that variable. Destructive functions include

(RPLACA ⟨variable⟩ ⟨expression⟩)
(RPLACD ⟨variable⟩ ⟨expression⟩)
(DREMOVE ??)
(SORT ??)

(i.e., see Section 3.9 and consider any destructive functions).

```
←. WHO IS SMASHED BY WHOM
MODEL.NAME -- (DELETE.MODEL)
X -- (REPLACA)
done
```

TEST

A variable is tested by a function if its value is only distinguished between NIL and non-NIL values. Testing expressions include

(COND (⟨variable⟩ ...)
(AND ⟨variable⟩ ...)

```
←. WHO IS TESTED BY WHOM
CURRENT.DATABASE.NAME -- (MAKEDEMON INHERIT MAKENODE
EXISTP ADD.TO.DB.VARS OPEN.AND.LOAD.DATABASE
INITIALIZE.NEW.DATABASE)
INSTANCE -- (CREATE.INSTANCE)
SLOTS -- (INITIALIZE.SLOTS)
etc...
```

REFERENCE	A variable is referenced by a function if its used in any other way than being set.
USE	If unmodified, a function uses a variable if it satisfies any of SET, SMASH, TEST, or REFERENCE.

```
←. WHO USES A.DB.GRAPH FREELY
(SHOW.DATA.MODELS SHOW.OPEN.DATABASE.STRUCTURESHOW.
DATABASE.MODELS)

← WHO IS USED FREELY BY ANY CALLING 'EXISTP
(NODE)
```

BIND	A variable is bound by a function if it occurs in the argument list of the function or a PROG expression within the function.
USE AS A FIELD	A name is used as a field when it occurs as a record field name in create, fetch, or replace expressions.
FETCH	A name appears as a field name within a fetch expression.
REPLACE	A name appears as a field name within a replace expression.
CREATE	A name appears as a record name within a create expression.
USE AS A RECORD	A name is used as a record name in create or TYPE? expressions.
USE AS A PROPERTY NAME	A name is used as a property name in one or more property list functions. These include expressions of the form (GETPROP ⟨atom⟩ ⟨name⟩) (PUTPROP ⟨atom⟩ ⟨name⟩⟨expression⟩) etc.
USE AS A CLISP WORD	A name is either an iterative statement operator or a user defined CLISP word.

CONTAIN	Functions, records, and variables are contained within files. Masterscope computes this relation dynamically from File Package information.
DECLARE AS LOCALVAR DECLARE AS SPECVAR	Variables may be declared as local variables or special variables within a function.

Modifying Verbs

The verbs USE, SET, SMASH, and REFERENCE may be modified by the adverbs FREELY and LOCALLY. These adverbs determines whether a variable is bound within the stack frame of the function.

```
←. WHO IS USED LOCALLY
(TEMP TRUNCATION SIZE CX3 R CX1 CX2 I CX Y X)
```

26.5 SPECIFYING PATHS

A *path* is a specification through the hierarchy of function calls. SHOW PATHS takes a path specification to determine what functions it should identify for you. There are several path options that you may specify:

FROM ⟨set⟩ — This option displays all function calls from the elements of ⟨set⟩.

```
←. SHOW PATHS FROM CDIFFERENCE
1.CDIFFERENCE COMPLEX
2.            REAL
3.            IMAG
NIL
```

TO ⟨set⟩ — This option displays the function calls that lead to elements of ⟨set⟩. If TO is given before FROM, the tree will be inverted and a warning message will be printed.

When both FROM and TO are specified, Masterscope traces the elements of the set given by

```
CALLED SOMEHOW BY X AND CALLING Y
SOMEHOW
```

If TO is not specified, the following form is assumed:

TO KNOWN OR NOT @ GETD

```
←. SHOW PATHS TO REAL
(inverted tree)
1.REAL    CPLUS
2.        CDIFFERENCE
3.        CMULT
4.        PRINT.COMPLEX
NIL
```

AVOIDING ⟨set⟩ — This option does not display any function that is an element of ⟨set⟩. AMONG is a synonym for AVOIDING NOT.

```
←. SHOW PATHS TO REAL AVOIDING CMULT
(inverted tree)
1.REAL    CPLUS
2.        CDIFFERENCE
3.        PRINT.COMPLEX
NIL
```

NOTRACE ⟨set⟩ — This option does not trace any element from ⟨set⟩. Functions marked NOTRACE are printed in the tree, but the tree is not expanded beyond that function.

SEPARATE ⟨set⟩ — This option displays a separate tree for each element of ⟨set⟩. A tree may become very complex when you have many layers in the function call hierarchy. This makes it difficult to comprehend the paths to various functions. Judicious use of the SEPARATE option will make the function hierarchies displayed by SHOW PATHS more comprehensible.

LINELENGTH ⟨n⟩ — This option resets the line length of the output file before displaying the tree. The line length is used to determine when the tree should

overflow and, therefore, be expanded at a lower level.

26.6 DESCRIBING FUNCTION BEHAVIOR

The behavior of a function is described by a *template*. A template is a pattern of a function's evaluation. It is a list consisting of a set of atoms that describe each of the arguments to the function.

Masterscope templates are stored in a hash array whose address is the value of MSTEMPLATES. To inspect this array, use EDITV on the value of MSTEMPLATES.

The atoms which may appear in a template are

PPE — Indicates a parenthesis error when an expression appears in this location.

```
CDR        (NIL EVAL . PPE)
```

means that the CDR is not evaluated, its argument is, and nothing more should appear in the argument list.

NIL — Indicates that an expression occurring at this location is not evaluated.

```
GO         (NIL NIL . PPE)
```

means that neither the GO nor its label are evaluated and, moreover, nothing else may occur in the expression.

SET — A variable appearing at this location is set by the function.

```
SAVESETQ        (NIL SET EVAL . PPE)
```

means that the first argument is set to the value of the second argument which is evaluated.

SMASH — Indicates that the value of the expression appearing at the location is destructively modified in place.

```
/LCONC     (CALL SMASH EVAL . PPE)
```

TEST — The expression appearing at this location is used as a predicate. The argument at this location is not evaluated.

```
GO          (NIL NIL . PPE)
```

means that neither the GO nor its label are evaluated and, moreover, nothing else may occur in the expression.

RETURN — The value of the function is given by the value of this expression.

```
AND         (NIL .. TEST RETURN)
```

means that the value of AND is the value of its last argument if all of its arguments evaluate true.

EFFECT — The expression appearing at this location is evaluated, but its value is not used by the function.

```
SELECTQ         (CALL EVAL .. (NIL ..
                EFFECT RETURN) RETURN)
```

means that each expression in the action part of a selector is evaluated, but only the last action is returned as the value of the selector.
Expressions represented by EFFECT may affect other variables; they may be viewed as side effects.

FETCH — An atom at this location is the name of a field that is to be fetched.

REPLACE — An atom at this location is the name of a field that is to be replaced.

RECORD — An atom at this location is the name of a record.

CREATE — An atom at this location is the name of a record that is to be created.

BIND — An atom at this location is a variable that is bound.

```
PROG      (! NIL
             (BOTH
               (.. (IF LISTP
                       (NIL EVAL ..
                       EFFECT)
                       NIL))
               (.. (IF LISTP
                       (BIND BIND)))
             .. (IF LISTP EFFECT))
```

CALL — An atom at this location is a function that is called.

```
ERRORSET (CALL EVALQT .. EVAL)
```

Note that functions which have NIL as their first atom are executed directly by the interpreter, whereas those with CALL as their first atom are defined in terms of the more primitive functions.

CLISP — An atom at this location is used as a CLISP word.

TESTRETURN — A combination of TEST and RETURN. If the value of the function is non-NIL, then it is returned. For instance, a one-element COND clause is analyzed this way.

```
OR        (NIL .. TESTRETURN RETURN)
```

These atoms are supplemented by a set of special forms. Each form is a list whose CAR is interpreted according to the above atoms. These forms are

.. ⟨template⟩ — When .. appears before a part of the template, the next element of the template may occur an indefinite number of times.

EVAL — (CALL EVALQT .. EVAL)

RESETFORM — (CALL .. EVAL)

PROG1 — (NIL RETURN .. EFFECT)

(BOTH ⟨template⟩ ⟨template⟩) Indicates that the current expression is to be analyzed twice using each of the templates in turn.

PROG

```
(NIL
  (BOTH
    (..
      (IF LISTP
        (NIL EVAL ..
        EFFECT) NIL))
    (..
      (IF LISTP (BIND)
      BIND)))
  .. (IF LISTP EFFECT))
```

(IF ⟨expr⟩ ⟨temp⟩ ⟨temp⟩) The expression is evaluated when the function is analyzed by Masterscope. If the result is non-NIL, use the first template to analyze the function; otherwise, use the second template.

EVALA

```
(BOTH (IF & &) (CALL EVALQT
EVAL . PPE)
```

(@ ⟨exprform⟩ ⟨tempform⟩) The ⟨exprform⟩ is evaluated to produce an ⟨expr⟩, and the ⟨tempform⟩ is evaluated to give a ⟨template⟩, both when the function is analyzed. The ⟨expr⟩ is analyzed using ⟨template⟩. This allows you to dynamically analyze expressions.

DECLARE

```
(CALL .. (@ EXPR &))
```

(MACRO . ⟨macro⟩) The form ⟨macro⟩ is interpreted like a compiler macro. The resulting form is analyzed.

```
RESETVARS                    MACRO
```

Some of the templates may be rather complex. Consider the following examples:

```
DECLARE          (CALL ..
                   (@ EXPR
                      (CONS NIL
                            (SELECTQ (CAR (LISTP EXPR))
                                     (LOCALVARS
                                        (QUOTE (IF LISTP
                                                   (..
                                                   LOCALVARS)
                                                   LOCALVARS)))
                                     (SPECVARS
                                        (QUOTE (IF LISTP
                                                   (..
                                                   SPECVARS)
                                                   SPECVARS)))
                                     NIL))))

SETATOMVAL       (CALL (IF (EQ (CAR (LISTP EXPR))
                               (QUOTE QUOTE))
                           (NIL SET)
                           EVAL)
                     EVAL . PPE)

EVALA            (BOTH (IF (EQ (CAR (LISTP (CADDR EXPR)))
                               (QUOTE QUOTE))
                           (NIL NIL (NIL (.. (BIND)))))
                     (CALLEVALQT EVAL . PPE))

FUNCTION         (CALL (REMOTE (IF LITATOM CALL LAMBDA))
                  (IF LITATOM EVAL NIL))
*                (! (IF (EQ (CADR EXPR)
                            (QUOTE ASSERT:))
                       (NIL NIL
                            ..
                            (IF LISTP
                                (@ (CDR EXPR)
                                   (LIST QUOTE ..)
                                   (MSWORDNAME (CAR
                                   EXPR)))))))
```

Masterscope templates are stored in a hash array whose address is stored in the variable MSTEMPLATES. You may inspect the individual entries in the hash array by performing the following operations (in INTERLISP-D):

```
←DV[MSTEMPLATES]
```

which opens a DEDit window displaying the HARRAYP address. Then, select EDITV on the HARRAYP address to open another window displaying the individual entries.

26.7 MASTERSCOPE FUNCTIONS

The Masterscope database is updated by many of the INTERLISP subsystems. Updating occurs through function calls that add information to or modify information in the database. Several functions allow you to use the Masterscope database from within your programs.

26.7.1 Entering Masterscope

You enter Masterscope from the INTERLISP top-level READ-EVAL-PRINT loop by executing **MASTERSCOPE**, which takes the form

Function:	MASTERSCOPE
# Arguments:	1
Argument:	1) a command, COMMAND
Value:	The value returned by executing the command.

If COMMAND is NIL, Masterscope enters into a user executive (see Section 25.3) through which you may enter commands to be executed. Otherwise, Masterscope attempts to execute COMMAND and returns the value, if meaningful, that is its result.

```
←(MASTERSCOPE)
Masterscope 4-AUG-83... Type HELP⟨cr⟩ for command summary.
```

A common call to Masterscope is to erase the database when loading a new application system. Thus, you might enter the following function call into your File Package commands:

```
(MASTERSCOPE 'ERASE)
```

26.7.2 Determining Who a Function Calls

CALLS extracts an analysis of a function from the database. It takes the form

```
Function:       CALLS
# Arguments:    2
Arguments:      1) a function name, FN
                2) a database, USEDATABASE
Value:          As described below.
```

CALLS returns a list consisting of four segments of information:

1. a list of functions called by FN
2. a list of variables bound in FN
3. a list of variables used freely in FN
4. a list of variables used globally in FN.

FN may be a function name, a definition (for example, a LAMBDA expression), or an S-expression.

If USEDATABASE is non-NIL (usually T), and FN is a function name, CALLS uses the Masterscope database. If FN is not a literal atom (e.g., a function name) or USEDATABASE is NIL, CALLS performs an analysis of the expression.

```
←. (CALLS 'MAKESLOT 'T)
((EQUAL PUTPROP MAPCAR EXISTP ENTER GETPROP
LIST GIVE.TO.OFFSPRING)
(INHERITANCE.TYPE METHOD NODE SLOT)
(〈NIL〉))
```

CALLSCCODE

An alternative form of CALLS, **CALLSCCODE**, analyzes compile code (remember: we said Masterscope only works on interpreted code). CALLSCCODE analyzes compiled code and returns a list consisting of five elements:

1. a list of functions called via "linked" function calls
2. a list of functions called regularly
3. a list of variables bound in FN
4. a list of variables used freely
5. a list of variables used globally.

CALLSCCODE takes the form

```
Function:       CALLSCCODE
# Arguments:    1
Argument:       1) a function name, FN
Value:          As described above.
```

26.7.3 Determining the Free Variables

You may determine the free variables used within a function by executing **FREEVARS**, which takes the form

```
Function:       FREEVARS
# Arguments:    2
Arguments:      1) a function name, FN
                2) a database, USEDATABASE
Value:          A list of the free variables used within
                FN.
```

FN may be a function name, a definition, or a form. If USEDATABASE is NIL or FN is an S-expression, FREEVARS performs an analysis of the expression to determine the free variables. Otherwise, USEDATABASE is usually T, and the free variables are determined from the Masterscope database. In this case, if FN has not been previously analyzed, it is now analyzed and the information entered into the Masterscope database.

Consider the following example:

```
←(FREEVARS 'SHOW.DATA.MODEL)
(NODE DATABASE.WINDOW A.DB.MODEL A.DB.GRAPH)
```

26.7.4 Getting and Setting Templates

Masterscope uses templates to analyze a function behavior. It knows about many of the basic functions provided by INTERLISP. You may obtain a template by executing **GETTEMPLATE**, which takes the form

```
Function:       GETTEMPLATE
# Arguments:    1
Argument:       1) a function name, FN
Value:          The template used by Masterscope, if
                available.
```

Consider the following examples:

```
←(GETTEMPLATE 'TCONC)
(SMASH EVAL . PPE)

←(GETTEMPLATE 'SAVESET)
((IF
     (EQ (CAR (LISTP EXPR)) (QUOTE QUOTE))
     (NIL SET) EVAL) EVAL EVAL EVAL . PPE)

←(GETTEMPLATE 'RPTQ)
(EVAL (BOTH (@ (QUOTE RPTN) (QUOTE BIND)) RETURN) . PPE)
```

You may establish a template for a function using **SETTEMPLATE**, which takes the form

```
Function:       SETTEMPLATE

# Arguments:    2

Arguments:      1) a function name, FN
                2) an S-expression, TEMPLATE

Value:          The old template, if any, for FN.
```

SETTEMPLATE changes the template for FN in MSTEMPLATES. It returns the old template, if any.

26.7.5 Defining Masterscope Synonyms

Masterscope responds to many different commands. You may define synonyms for frequently used commands using **SETSYNONYM**, which takes the following form

```
Function:       SETSYNONYM

# Arguments:    2

Arguments:      1) a new phrase, NEWPHRASE
                2) a meaning for the phrase, MEANING

Value:          NIL.
```

Masterscope defines a relationship between NEWPHRASE and MEANING. Whenever it sees NEWPHRASE in a command, it substitutes MEANING for NEWPHRASE before executing the command. Consider the following example (after the IRM):

```
←(SETSYNONYM 'GLOBALS
             '(VARS IN GLOBALVARS
              OR @(GETPROP X 'GLOBALVAR)))
NIL
```

26.7.6 Parsing a Relation

Masterscope provides several functions for users who want to write their own analysis routines. **PARSERELATION** creates an internal representation (e.g., a subset of the Masterscope database structure) for the given relation. It takes the form

```
Function:       PARSERELATION
# Arguments:    1
Argument:       1) a relation, RELATION
Value:          The internal representation.
```

Relations are described in Section 26.4. The internal representation returned by PARSERELATION is not meaningful to you. Usually, you set its value to a variable which is later given to GETRELATION. Consider the following example:

```
←(PARSERELATION '(USE FREELY))
(TABLES
(({HARRAYP}#7,1624 . MSREHASH)
     {HARRAYP}#7,1020 . MSREHASH)
(({HARRAYP}#1,2574 . MSREHASH)
     {HARRAYP}#1,2340 . MSREHASH)
(({HARRAYP}#7,1110 . MSREHASH)
     {HARRAYP}#7,1114 . MSREHASH)
(({HARRAYP}#1,2054 . MSREHASH)
     {HARRAYP}#7,1204 . MSREHASH))
```

Primitive relations are stored in a pair of hash tables, one for the "forward" direction, and one for the "reverse" direction. Hash tables take less storage space. Searching is more efficient which means fast access. To retrieve information from the Masterscope database (or a subset of it), Masterscope performs a union across the relevant tables as determined by the relation.

26.7.7 Getting the Results of a Relation

GETRELATION evaluates an item with respect to the given relation and returns the result. In effect, PARSERELATION and GETRELATION allow you to cir-

cumvent the analysis of commands which are frequently used by analyzing them once. It takes the form

```
Function:       GETRELATION

# Arguments:    3

Arguments:      1) an item, ITEM
                2) a relation, RELATION
                3) an inversion flag, INVERTED

Value:          A list of atoms satisfying the relation.
```

RELATION should be an S-expression specifying a relation or an internal representation that was previously created by PARSERELATION. If RELATION is an S-expression, GETRELATION calls PARSERELATION to create the corresponding internal representation.

ITEM is an atom. GETRELATION analyzes ITEM with respect to RELATION and returns a list of atoms, if any, that satisfy the relationship. Consider the following example:

```
←(GETRELATION 'SHOW.DATA.MODEL '(USE FREELY))
SLOT     -   " "  -   GIVE.TO.OFFSPRING DO.SUBNODE
                      DO.DEMON
FROM     -   " "  -   INHERIT.ALL.SLOTS.OR.DEMONS
                      INHERIT.SLOT
TO       -   " "  -   INHERIT.ALL.SLOTS.OR.DEMONS
etc...
```

If INVERTED is T, then the sense of the relation is inverted. That is, instead of determining who SHOW.DATA.MODEL uses freely, GETRELATION determines who uses SHOW.DATA.MODEL freely.

If ITEM is NIL, GETRELATION returns a list of atoms which have the relation with any other atom, i.e., it answers the question "WHO ⟨relation⟩S ANY".

27.7.8 Testing a Relation

TESTRELATION allows you to test if an item has a relation to another item. It takes the form

```
Function:       TESTRELATION

# Arguments:    4

Arguments:      1) an item, ITEM
                2) a relation, RELATION
```

```
                3) an item to be tested, ITEM2
                4) an inversion flag, INVERTED
Value:          A list of functions satisfying the
                relation, if any.
```

TESTRELATION is equivalent to

```
(MEMBER ITEM2
        (GETRELATION ITEM RELATION INVERTED))
```

If ITEM2 is NIL, the function call is equivalent:

```
(NOT (NULL (GETRELATION (ITEM RELATION INVERTED))))
```

i.e., it tests if ITEM has the given relation with any other item.

26.7.9 Mapping Across a Relation

MAPRELATION applies a function to every pair of items related via a specified relation. It takes the form

```
Function:       MAPRELATION
# Arguments:    2
Arguments:      1) a relation, RELATION
                2) a mapping function, MAPFN
Value:          The result of applying MAPFN to the pairs
                of items.
```

26.7.10 Updating the Database

You may update a function that you have created or modified using **UPDATEFN**, which takes the form

```
Function:       UPDATEFN
# Arguments:    2
Arguments:      1) a function, FN
                2) an analysis flag, EVENIFVALID
Value:          NIL.
```

This function operates like the command ANALYZE ⟨FN⟩. UPDATEFN will analyze FN if it has not previously been analyzed or if it has been changed

since it was last analyzed. If EVENIFVALID is T, UPDATEFN will reanalyze FN even if Masterscope thinks it has a valid analysis in the database.

```
←(UPDATEFN 'CMULT T)
.NIL
```

UPDATECHANGED, which takes no arguments, applies UPDATEFN to every function which has been marked as changed. It takes the form

```
Function:       UPDATECHANGED
# Arguments:    0
Arguments:      N/A
Value:          NIL.
```

MSMARKCHANGED marks a function, FN, as changed and notes that it needs to be reanalyzed. It takes the form

```
Function:       MSMARKCHANGED
# Arguments:    3
Arguments:      1) a function, FN
                2) a type, TYPE
                3) a reason code, REASON
Value:          NIL.
```

The values of REASON are the same as those used by the function MARKASCHANGED (see Section 17.3.9).

26.7.11 Dumping the Masterscope Database

DUMPDATABASE dumps the current Masterscope database on the current output file in a loadable form. It takes the form

```
Function:       DUMPDATABASE
# Arguments:    1
Argument:       1) a list of functions, FNSLST
Value:          NIL.
```

If FNSLST is non-NIL, DUMPDATABASE will only dump the information for those functions which appear in FNSLST.

```
←(DUMPDATABASE 'CMULT)
(READDATABASE)
(
CALL CMULT (COMPLEX DIFFERENCE TIMES REAL IMAG PLUS) NIL
BIND CMULT NIL NIL
NLAMBDA CMULT NIL NIL
NOBIND CMULT NIL NIL
RECORD CMULT NIL NIL
CREATE CMULT NIL NIL
FETCH CMULT NIL NIL
REPLACE CMULT NIL NIL
REFFREE CMULT NIL NIL
REF CMULT (CX1 CX2) NIL
SETFREE CMULT NIL NIL
SET CMULT (CX3) NIL
SMASHFREE CMULT NIL NIL
SMASH CMULT NIL NIL
PROP CMULT NIL NIL
TEST CMULT NIL NIL
TESTFREE CMULT NIL NIL
PREDICATE CMULT NIL NIL
EFFECT CMULT NIL NIL
...
)
NIL
```

where the listing provides a description of CMULT from all the possible ways Masterscope analyzes a function.

A variable, DATABASECOMS, is initialized to

```
((E (DUMPDATABASE)))
```

Thus, you may execute a MAKEFILE as follows:

```
(MAKEFILE 'DATABASE.⟨extension⟩)
```

to save the current Masterscope database.

27

The Record Package

The *Record Package* permits you to define data structures whose usage is independent of structural details. You declare the data structures to be used by your programs through *record* declarations. Thereafter, you can manipulate the data without concern for the underlying data structures. INTERLISP automatically computes the appropriate expressions from the data structure declarations that are necessary to accomplish the access and storage operations. You may change the data structure merely by changing the declarations. The program automatically adjusts itself to these new conventions.

27.1 RECORD DECLARATIONS

A *record declaration* specifies the format of a data structure known as a *record*. Each record has a name that identifies the data structure. Each record is composed of a variable number of *fields*. The specification of the record's fields depends on the type of the record. There are currently ten record types defined by INTERLISP. Fields are identified by names. For each field, INTERLISP generates the expression necessary to retrieve data from or store data to that field when it is referenced by the program. Fields may be further subdivided into *subfields* by subdeclarations. In effect, records may recursively contain other record declarations.

The Record Package is implemented via the CLISP and DWIM subsystems. Record operations are translated using CLISP declarations. You may declare local record declarations to override global ones.

27.1.1 Components of the Record Declaration

There are ten record types recognized by the Record Package. A record type implicitly specifies how data are accessed or stored within the record (i.e., the "data path"). The record types are given below:

Record Types

Record Type	*Data Structure Supported*
RECORD	generic list
TYPERECORD	"typed" list
DATATYPE	user-defined datatypes
ARRAYRECORD	arrays
ATOMRECORD	property list
PROPRECORD	lists with property list format
ASSOCRECORD	association lists
HASHLINK	typeless, but hash-linked
ACCESSFNS	typeless, but access is defined in the record declaration itself

The *record name* is a literal atom that identifies the data structure. The record name is used by the Record Package functions, by the File Package, and to specify a template for an arbitrary list structure.

The *fields* of a record are the description of its structure. Fields may or may not have names. Those with names are accessible by the program. NIL is used to indicate those fields that the program will not access. This facility allows you to apply several different templates to a single data structure. Each template has a definition for the fields to be accessed when that template is applied to the data structure. This mechanism is not unlike the concept of a "view" or a "subschema" that is used in database management systems. The interpretation of the "field" component depends on the record type. These interpretations will be addressed in the sections describing each record type (see Sections 27.4 through 27.9).

Finally, a record declaration may be terminated by a *record tail*. Record tails are optional components of the declaration. The record tail allows the user some control over the manner in which certain Record Package functions are applied to the record.

The format for a record declaration is written as follows:

```
(record-type record-name fields [record-tail])
```

27.1.2 Using the Record Declaration

To use a record structure, you prefix a field name from the record with the name of a literal atom whose value is a data structure corresponding to the record type. Let us assume we have defined the following record:

```
←(RECORD president
    (     first.name NIL election.year party votes
          loser.name loser.party . loser.votes))
PRESIDENT
```

Note that the period is significant and required before the last field name by the declaration.

Let us further assume that we have two records (note lower case, indicating instances of the template):

```
←(SETQ kennedy
      '(john f 1960 democrat 303 nixon whig 219))
(JOHN F 1960 DEMOCRAT 303 NIXON WHIG 219)

←(SETQ nixon
    '(richard m 1972 republican 521 mcgovern democrat 17))
(RICHARD M 1972 REPUBLICAN 521 MCGOVERN DEMOCRAT 17)
```

We can obtain the values of various fields as follows:

```
←kennedy:election.year
1960

←nixon:loser.name
mcgovern
```

When you write a statement of the form "kennedy:election.year", you are implicitly saying that "kennedy" is an instance of the record PRESIDENT. The interpretation of "kennedy:⟨field name⟩" depends on the current value of KENNEDY.

What happens, then, if the underlaying data structure is not equivalent to the template that is applied to it? Depending on the operation to be performed, you will receive NIL, some unexpected result, or an error message. The Record Package does not provide any facilities for run-time validation of data structures nor is there any error checking other than that provided by INTERLISP itself.

We can also use the components of records as arguments to any INTERLISP function.

"Was Kennedy's margin of victory greater than Nixon's?" might be encoded as

```
←(GREATERP
     (DIFFERENCE kennedy:votes kennedy:loser.votes)
     (DIFFERENCE nixon:votes   nixon:loser.votes))
NIL
```

Clearly, by inspecting the records above, we see that the answer is false (represented by NIL).

Now, in the record giving information about "kennedy", we have made a slight error. Richard Nixon did not belong to the Whig party, but to the Republicans. We can replace the value of the field "loser.party" in the record "kennedy" as follows:

```
kennedy:loser.party←republican
```

which translates to

```
(CAR (RPLACA (CADR (CDDDDR kennedy)) 'republican))
```

The value returned from a storage operation is the value that is assigned to the field of the record, unless an error occurs.

27.1.3 Translating the Record Declaration

The record declaration is translated into a set of record operations that are then stored elsewhere by CLISP. For example, the field reference "kennedy:election.year" is equivalent to

```
(CADDR kennedy)
```

INTERLISP replaces each access to a record field by an expression of the form

```
(fetch ⟨field name⟩ of ⟨atom⟩)
```

Thus, we would see "kennedy:election.year" expressed as

```
(fetch election.year of kennedy)
```

INTERLISP replaces each store to a record field by an expression of the form

```
(replace ⟨field name⟩ of ⟨atom⟩ by ⟨value⟩)
```

Thus, we would see "kennedy:loser.party←republican" expressed as

```
(replace loser.party of kennedy by republican)
```

27.1.4 Record Subfields

As we mentioned above, record declarations may be recursively nested to yield subdeclarations of individual fields. In every election, we have one winner, but we may have more than one loser (as has happened in several elections). Thus, the loser field needs to be replicated to represent this information. We can describe this by declaring the loser field to be a RECORD itself. Let us assume there are only two losers per election. Then, we can redefine PRESIDENT as follows:

```
←(RECORD PRESIDENT
   (first.name NIL election.year party votes . loser)
   (RECORD loser
        (loser.name1 loser.party1 loser.votes1
         loser.name2 loser.party2 . loser.votes2)))
PRESIDENT
```

We might specify the election record for Ronald Reagan as follows:

```
(SETQ reagan
           (CREATE presidents
                   first.name←ronald
                   election.year←1980
                   party←republican
                   votes←465
                   loser.name1←carter
                   loser.party1←democrat
                   loser.votes1←70
                   loser.name2←anderson
                   loser.party2←independent
                   loser.votes2←0))
```

27.2 CREATING A RECORD

As we mentioned above, record operations may be applied to any arbitrary data structure because the record declaration acts as a template that overlays the data structure to define its fields. However, caution should be exercised because a mismatch between the template and the actual data structure may result in an error. A good policy to follow is to create new instances of records using the record declarations that define their access and storage paths.

To create a new instance of a record, the Record Package provides the **CREATE** expression, which takes the form

```
(CREATE record-name . ⟨assignments⟩)
```

and is usually used in a SETQ statement as follows:

```
(SETQ ⟨atom⟩ (CREATE record-name ⟨assignments⟩)
```

which gives the atom a value consisting of the data structure described by the record declaration. INTERLISP initializes the data structure according to the specified record type.

The [assignments] field is optional. It specifies the methods by which individual elements of the data structure are to be initialized. Elements that are not specified in [assignments] will be given the default value, if one is specified in the record-tail or NIL.

Expressions in ⟨assignments⟩ may take one of the following forms:

field-name← S-expression
The expression initializes the value of the field to the value of the S-expression when it is evaluated.

```
←(SETQ johnson
    (CREATE presidents
            first-name←lyndon
            election-year←1964
            party←democrat
            votes←486))
(lyndon 1964 democrat 486)
```

initializes a segment of the record that describes Lyndon Johnson's election.

USING S-expressions
This expression places higher priority on the presence of a default in the record tail. For each field name, if there is no value specified in an expression in the record tail, then the value defined in the USING expression will be used to initialize that field. Let us assume the record declaration

```
←(RECORD state
      (admin.number population . electoral.votes) .
      electoral.votes←3)
STATE
```

which defines "state" as a record. The record tail specifies that each state has at least three electoral votes: two senators and at least one representative. Let us create a specific state:

```
←(SETQ california
    (CREATE state
            admin.number←18
```

```
            USING (19953134 45)))
(18 19953134 45)
```

which is translated as

```
(LIST 18 (CADR '(19953134 45)) (CADDR '(19953134 45)))
```

We could also set the S-expression as the value of an atom and place the atom's name in the USING expression as follows:

```
←(SETQ newdata '(19953134 45))
(19954134 45)
←(SETQ california
     (CREATE state admin.number←18 USING newdata))
(18 19953134 45)
```

In either case, because we have explicitly specified "admin.number", the value is taken from the CREATE expression. However, the value of "electoral.votes" remains 3 because there is a default specified in the record tail.

COPYING S-expressions

This expression is like USING except that the corresponding values are copied from the S-expression using COPYALL (see Section 6.4.2). The example above would translate as follows:

```
←(SETQ california
(CREATE state
       admin.number←18
       COPYING newdata))
(18 19953134 45)
```

becomes

```
(LIST 18 (COPYALL (CADR newdata)) (COPYALL (CADDR
newdata)))
```

REUSING S-expressions

This expression is like USING except that, wherever possible, the corresponding structure in the S-expression is used:

```
←(SETQ ca
   (CREATE state
          admin.number←18
          REUSING newdata))
(18 19953134 45)
```

The difference between USING and REUSING is that CREATE will attempt to incorporate as much of the old data structure into the new data structure being created. The USING clause creates an entirely new data structure.

SMASHING S-expressions
This expression does not create a new instance of the record; rather, it uses the value of the S-expression and overlays the record structure on the S-expression itself.

The Definition of CREATE
CREATE is not defined as a function. Rather, DWIM recognizes CREATE as a keyword and calls a Record Package function with the entire CREATE expression as an argument. The definition of CREATE is found by

```
←(GETPROPLIST 'CREATE)
(CLISPWORD (RECORDTRAN . create))
```

The translation of CREATE is found by

```
←(DWIMIFY '(CREATE STATE USING NEWDATA))
(CONS (CAR NEWDATA)
      (CONS (CAR (CDR NEWDATA))
            (CDR (CDR NEWDATA))))
```

27.3 TESTING FOR RECORDS

Record templates cannot be applied indiscriminately to any data structure because an error may result. Thus, INTERLISP provides a facility for testing to see if a data structure "looks like" an instance of a particular record.

To test for a record, you use the **TYPE?** expression, which takes the following form:

```
(TYPE? record-name S-expression)
```

where it attempts to determine if the S-expression has the same form as record name. TYPE? performs differently according to the record type it is checking:

DATATYPE
TYPE? returns non-NIL if and only if the S-expression is an instance of the record name; type checking is exact.

TYPERECORD
TYPE? checks to see that the S-expression is a list beginning with record name. Note that TYPERECORDs are identified in their data structure by the name of the record declaration.

ARRAYRECORD
TYPE? checks to see that the S-expression is an array with the correct size.

PROPRECORD
TYPE? checks to see that the S-expression is a property list with property names chosen from those specified in the field component of the record declaration.

ASSOCRECORD
TYPE? checks to see that the S-expression is an association list with property names chosen from the field component of the record declaration.

TYPE? may not be applied to record instances of the types RECORD, ACCESSFNS, or HASHLINK. In each case, it will produce the error message TYPE? NOT IMPLEMENTED FOR THIS RECORD.

In Chapter 11, we defined a new record type, COMPLEX. We may see how TYPE? is used as follows:

```
←(SETQ CX1 (COMPLEX 1.0 3.0))
((1.0 . 3.0))
←(TYPE? COMPLEX CX1)
T
←(SETQ CX2 '(1.0 . 3.0))
(1.0 . 3.0)
←(TYPE? COMPLEX CX2)
NIL
```

The Translation of TYPE?
The translation of TYPE? is found by

```
←(DWIMIFY '(TYPE? COMPLEX CX1))
(TYPENAMEP CX1 (QUOTE COMPLEX))
```

27.4 MANIPULATING RECORD FIELDS

You may wish to manipulate the values of the fields of a particular record. The **WITH** construct may be used to manipulate record fields as if they were variables. It takes the form

```
(WITH record-name record-instance ⟨form1⟩ ... ⟨formN⟩)
```

where RECORD-NAME is the name of the record and RECORD-INSTANCE is an expression evaluating to a specific instance of that record. The forms are evaluated so that references to variables which are actually field names of the record are treated via FETCH and REPLACE.

```
←(RECORD RECN (FLD1 . FLD2))
RECN

←(SETQ DATA (LIST 10 20))
(10 20)

←(SETQ INSTANCE
      (CREATE RECN USING DATA))
(10 20)

←(WITH RECN INSTANCE (SETQ FLD2 (PLUS FLD1 FLD2)))
NON-NUMERIC ARG
(20)
```

because the translation is not done properly (although the IRM claims otherwise!).

The translation of this form is

```
←(DWIMIFY
   '(WITH RECN INSTANCE (SETQ FLD2 (PLUS FLD1 FLD2))))
(CDR (RPLACD INSTANCE (PLUS (CAR INSTANCE)
                            (CDR INSTANCE))))
```

However, the following expression will operate correctly:

```
←(WITH RECN INSTANCE (SETQ FLD2 (PLUS FLD1 30)))
40

←INSTANCE
(10 . 40)
```

but the correction for the problem above is merely to preface the field with the record name as follows:

```
←(WITH RECN INSTANCE (SETQ FLD2 (PLUS FLD1
INSTANCE:FLD2)))
50

←INSTANCE
(10 . 50)
```

whose translation is still the same as given above.

27.5 RECORDS AND TYPED RECORDS

The Record Package provides two record types for declaring list structures: **RECORD** and **TYPERECORD**. The only difference between these two data structures is that typed records have the record name inserted as the first element.

We have already seen how the RECORD declaration works in the examples given in Section 27.1. Let us examine another example very briefly. Consider the record named ADMINISTRATION which is declared as follows:

```
←(RECORD administration
      (president-name inaugural-date . vice-president))
ADMINISTRATION
```

For both RECORDs and TYPERECORDs, the field descriptions are included within a single list. Let us create an instance of an ADMINISTRATION:

```
←(SETQ admin-1
    (CREATE administration
            (     president-name←washington
                  inaugural-date←04/30/1789
                  vice-president←adams)))
(WASHINGTON 04/30/1789 ADAMS)
```

while, for a TYPERECORD declaration, it would appear as

```
(ADMINISTRATION WASHINGTON 04/30/1789 ADAMS)
```

The translation of the access and storage operations takes the additional field into account, as follows:

```
←ADMIN-1:INAUGURAL-DATE
04/30/1789
```

would be translated as (CADR admin-1) for a RECORD declaration; it would be translated as (CADDR admin-1) for a TYPERECORD declaration.

Within a field specification, NIL may appear as a placeholder to indicate fields that you do not wish to reference through this declaration. We could describe the PRESIDENT record as follows:

```
←(RECORD president
      (birth-date death-date NIL spouse . state))
PRESIDENT
```

An actual instance of PRESIDENT (for Gerald Ford) might look something like

```
←(SETQ ford
   (07/14/1913 NIL republican elizabeth michigan))
(07/14/1913 NIL REPUBLICAN ELIZABETH MICHIGAN)
```

The field that we cannot access in this data structure is the name of Ford's party (i.e., Republican).

In any type of record declaration, NIL is used to indicate a "don't care" field. If several such fields occur consecutively, you may use an integer in the field description to indicate the number of consecutive NILs.

27.6 PROPERTY AND ASSOCIATION LIST RECORDS

A **PROPRECORD** has the format of a property list. FIELDS is a list of property names. The field names and their values are stored in a property list format:

```
(⟨fieldname⟩ ⟨value⟩ ... ⟨fieldname⟩ ⟨value⟩)
```

Field values are accessed via LISTGET and stored via LISTPUT (see Section 6.11).

When you create a PROPRECORD, only the fields that are actually named in the CREATE expression are stored in the list that represents the record. However, if you execute a CREATE expression with just the record name, the list representation consists of the first field name with a value of NIL.

An **ASSOCRECORD** is a record whose entries are stored in association list format. FIELDS is a list of literal atoms. The field names and their values are stored as follows:

```
((⟨fieldname⟩ . ⟨value⟩) ... (⟨fieldname⟩ . ⟨value⟩))
```

Accessing fields is performed via ASSOC while storing values into fields is performed via PUTASSOC.

Consider the following definitions (e.g., relations) which correspond to those often used in relational database examples:

```
←(PROPRECORD EMPLOYEE (EMPNAME AGE SALARY DEPNAME
MANAGER)) EMPLOYEE

←(PROPRECORD DEPARTMENT (DEPNAME FLOOR SALES DEPNO))
DEPARTMENT
```

We create an instance of a PROPRECORD using the CREATE expression:

```
←(CREATE EMPLOYEE EMPNAME←'SMITH
                  AGE←30
                  SALARY←25000
                  DEPNAME←'TOYS
                  MANAGER←'JONES)
(EMPNAME SMITH AGE 30 SALARY 25000 DEPNAME TOYS MANAGER
JONES)
```

```
←(CREATE DEPARTMENT DEPNAME←'TOYS)
(DEPNAME TOYS)
```

We can use an ASSOCRECORD to store information from a dictionary. Consider the following definition:

```
←(ASSOCRECORD DICTIONARY
              (POS LABEL DEFINITIONS ETYMOLOGY
              SYNONYMS))
DICTIONARY
←(SETQ JUNK
      (CREATE DICTIONARY
              POS←'NOUN
              DEFINITIONS←"1.0 Scrapped Materials
                          2.0 Chinese flat-bottomed
                              ship"
              ETYMOLOGY←'(PORTUGUESE JUNCO)))
((POS . NOUN)
 (DEFINITIONS . "1.0 Scrapped materials
                 2.0 Chinese flat-bottomed ship")
 (ETYMOLOGY PORTUGUESE JUNCO))
←JUNK:POS
NOUN
```

27.7 ARRAY AND HASHLINK RECORDS

An **ARRAYRECORD** associates field names with array elements. FIELDS is a list of atoms that are associated with the corresponding array elements. ARRAYRECORD only creates an array of pointers. Arrays of other datatypes must be created using ACCESSFNS which you write.

A **HASHLINK** sets up a record of hash arrays. FIELDS may take one of two values:

1. An atom, or
2. An S-expression having the following form:

```
(⟨fieldname⟩ ⟨arrayname⟩ ⟨arraysize⟩)
```

The ⟨arrayname⟩ indicates a hash array to be used. If it is not given, SYSHASHARRAY will be used (see Section 11.3). The ⟨arraysize⟩ is used to initialize the hash array if it has not previously been initialized. When the Record Package creates a hash array, it initializes via

```
(LIST (HARRAY (OR ⟨arraysize⟩ 100)))
```

An **ARRAYBLOCK** is a record of arrays. Its declaration form is similar to DATATYPE as described below. ARRAYBLOCK records are not supported in INTERLISP-D.

27.8 USER DATATYPE RECORDS

You may create new datatypes by declaring them to be records of the DATATYPE. FIELDS takes the form

```
(⟨fieldname⟩ . ⟨fieldtype⟩), or
⟨fieldname⟩
```

If ⟨fieldtype⟩ is omitted, the Record Package assumes a default type of POINTER.

The Record Package currently accepts the following specifications for ⟨fieldtype⟩:

POINTER	The field contains a pointer to any arbitrary INTERLISP object.
BITS ⟨n⟩	The field contains an ⟨n⟩-bit unsigned integer.
BETWEEN ⟨n1⟩ ⟨n2⟩	The field may contain an integer that lies in the range [⟨n1⟩ ⟨n2⟩]. That is, an integer, X, obeys the expression n1 ⟨= X ⟨= n2
INTEGER	The field contains a full-word signed FIXP integer. Note that the size of the word is machine-dependent.
FLOATING FLOATP	The field contains a floating point number.
FLAG	The field contains a one-bit field which indicates T or NIL.

User datatypes are created by executing an expression with DECLAREDATATYPE (see Section 27.12).

27.9 ACCESS FUNCTION RECORDS

An **ACCESSFNS** record is a record for which you may specify both an access and a storage function. INTERLISP captures the notion of access-oriented programming with this record type.

FIELDS is a list of entries of the form

```
(⟨fieldname⟩ ⟨access-definition⟩ ⟨store-definition⟩)
```

where, for each field name, you specify both an access and a storage description. These descriptions are usually function calls. The *access function* is a function of one argument, DATUM. It should return a value based on the value of DATUM. The storage function is a function of two arguments, DATUM and NEWVALUE. It should store NEWVALUE into DATUM and return NEWVALUE, if successful, as its value. If the storage definition is NIL or undefined, you will not be allowed to store data into the record field.

Consider the following example (after the IRM):

```
←(ACCESSFNS
   ((FIRSTCHAR (NTHCHAR DATUM 1)
               (RPLSTRING DATUM 1 NEWVALUE))
    (RESTCHARS (SUBSTRING DATUM 2))))
NIL

←(DWIMIFY '(REPLACE FIRSTCHAR OF X WITH Y))
(RPLSTRING X 1 Y)
```

Since no storage definition is given for the RESTCHARs field, attempting to perform a replacement here would cause an error:

```
←(DWIMIFY '(REPLACE RESTCHARS OF X WITH Y))
REPLACE not defined for this field
at RESTCHARS in (REPLACE RESCHARS OF X --)
```

Note that ACCESSFNS do not have a CREATE definition. You may specify one by the appropriate expression in the record tail.

The ACCESSFNS feature allows you to create data structures which cannot be specified by one of the other record types.

27.10 ATOM RECORDS

An *ATOMRECORD* returns an atom created by GENSYM (see Section 9.2.1) whose value is the record structure. FIELDS is a list of property names associated with an atom. Accessing is performed via GETPROP while storing is performed with PUTPROP. Consider the following example:

```
←(ATOMRECORD EMPLOYEE (ENAME AGE SALARY DNAME))
EMPLOYEE

←(SETQ EMPNO-1
      (CREATE EMPLOYEE ENAME←'SMITH
```

```
                                AGE←46
                                SALARAY←22000))
A0010
←(GETPROPLIST 'EMPNO-1)
(SALARY 22000 AGE 46 ENAME SMITH)
```

27.11 RECORD PACKAGE FUNCTIONS

The Record Package allows you to edit and manipulate record declarations from within programs. These functions are described in this section.

27.11.1 Editing a Record Declaration

EDITREC allows you to edit a global record declaration. It takes the form

```
Function:       EDITREC
# Arguments:    1-N
Arguments:      1) a record name, RECORDNAME
                2-N) optional editor commands, COM[1]  ...
                     COM[N]
Value:          The name of the record.
```

EDITREC is an NLAMBDA, nospread function. It invokes the editor on a copy of all declarations in which (CAR EDITRECX) is the record name or field name. When you exit the editor, all declarations that have been changed will be redeclared. All records that have been deleted will be undeclared. If (CAR EDITRECX) is NIL, all declarations will be edited.

27.11.2 Obtaining a Record Declaration

RECLOOK allows you to obtain the declaration for a given record name. It takes the form

```
Function:       RECLOOK
# Arguments:    1
Arguments:      1) a record name, RECORDNAME
Value:          The declaration for the record.
```

If RECORDNAME does not correspond to an existing declaration, RECLOOK returns NIL.

```
←(RECLOOK 'POSITION)
(RECORD POSITION (XCOORD YCOORD)
        (TYPE?
          (AND (LISTP DATUM))
          (NUMBERP (CAR DATUM))
          (NUMBERP (CDR DATUM))))

←(RECLOOK 'REGION)
(RECORD REGION (LEFT BOTTOM WIDTH HEIGHT)
        LEFT← -16383
        BOTTOM← -16383
        WIDTH← 32767
        HEIGHT← 32767
        (ACCESFNS
          ((TOP
              (IPLUS
                   (fetch (REGION BOTTOM) of DATUM)
                   (fetch (REGION HEIGHT) of DATUM)
                   -1))
           (PTOP
              (IPLUS
                   (fetch (REHION BOTTOM) of DATUM)
                   (fetch (REGION HEIGHT) of DATUM)))
           (RIGHT
              (IPLUS
                   (fetch (REGION LEFT) of DATUM)
                   (fetch (REGION WIDTH) of DATUM)
                   -1))
            (PRIGHT
              (IPLUS
                   (fetch (REGION LEFT) of DATUM)
                   (fetch (REGION WIDTH) of DATUM)))))
          (TYPE?
              (AND (EQLENGTH DATUM 4)
                   (EVERY DATUM (FUNCTION FIXP))))))
```

where both examples are taken from the Fugue release of INTERLISP-D.

27.11.3 Obtaining the Declarations of a Field

A field name may be used in many different record declarations. When you do so, you should ensure that the same semantics are attached to the field name in every instance. With this assumption in mind, **FIELDLOOK** returns a list of all declarations in which its argument is represented. Its single argument,

FIELDNAME, is the name of a field in zero or more declarations. It takes the form

```
Function:       FIELDLOOK
# Arguments:    1
Arguments:      1) a field name, FIELDNAME
Value:          The list of declaration in which FIELDNAME
                is the name of a FIELD.
```

Consider the following example:

```
←(FIELDLOOK 'REAL)
((DATATYPE COMPLEX ((REAL FLOATP) (IMAG FLOATP))))
```

27.11.4 Obtaining a Declaration's Field Names

Most records are declared global to the entire program, but declared in only one file that comprises the program. It may be necessary in other functions to know the names of the fields of a record. **RECORDFIELDNAMES** returns the names of the fields of a given record declaration. It takes the form

```
Function:       RECORDFIELDNAMES
# Arguments:    1
Arguments:      1) a record name, RECORDNAME
Value:          A list of the fields declared for the
                record.
```

Consider the following example:

```
←(RECORDFIELDNAMES 'PRESIDENT)
(LOSER.VOTES2 %. LOSER.PARTY2 LOSER.NAME2 LOSER.VOTES1
LOSER.PARTY1 LOSER.NAME1 LOSER %. VOTES PARTY
ELECTION.YEAR FIRST.NAME)
```

Note that the value returned does not distinguish between the fields of s subrecord and the primary fields of the record itself.

27.11.5 Accessing or Replacing a Record Value

RECORDACCESS allows you to access or replace a record value from within a function based on the name of the field. This facility is used when you do not

know the name of the field in advance, as when you ask the user to specify the field. It takes the form

```
Function:       RECORDACCESS

# Arguments:    5

Arguments:      1) a field name, FIELD
                2) an old value, VALUE
                3) an access mathod, TYPE
                4) a new value, NEW
                5) an (optional) declaration, DECL

Value:          The new value.
```

TYPE is one of the following methods:

FETCH or fetch
FFETCH or ffetch
REPLACE or replace
FREPLACE or freplace
/REPLACE or /replace

If TYPE is NIL, a default method of FETCH is assumed. Consider the following definition for COMPLEX (see Chapter 11):

```
←(PP COMPLEX)
(LAMBDA (R I)
     (PROG (TEMP)
           (SETQ TEMP (CREATE COMPLEX))
           (RECORDACCESS (QUOTE REAL)
                         TEMP NIL (QUOTE REPLACE)
                         R)
           (RECORDACCESS (QUOTE IMAG)
                         TEMP NIL (QUOTE REPLACE)
                         I)
           (RETURN TEMP)))
```

27.12 USER DEFINED DATATYPES

You may define new datatypes using **DECLAREDATATYPE**. This function allows you to define entirely new classes of objects, with a fixed number of fields specified by the definition of the data type. Additional functions allow you to define the name and type of fields for a given class, to create instances of objects of a given class, to access and replace the contents of individual fields within an object instance, and to interrogate objects about their contents.

27.12.1 Defining New Datatypes

You may define a new datatype using **DECLAREDATATYPE**, which takes the form

Function:	DECLAREDATATYPE
# Arguments:	2
Arguments:	1) a datatype name, TYPENAME 2) a field specification, FIELDS
Value:	A list of field descriptors that correspond to the elements of FIELDS.

TYPENAME is a literal atom which is the name of the new datatype. It may not conflict with existing datatypes defined by INTERLISP or the user.

FIELDS is a list of field specifications that describe the objects that are instances of the datatype. Each field specification takes the form

(⟨fieldname⟩ . ⟨fieldtype⟩)

The permissible field types are

POINTER	The field may contain a pointer to an INTERLISP object.
FIXP	The field contains an integer.
FLOATP	The field contains a floating point number.
(BITS ⟨n⟩)	The field contains a non-negative integer less than 2**N.

If FIELDS is NIL, TYPENAME is undeclared.

If TYPENAME is already declared, it will be redefined. That is, the datatype will be redeclared by INTERLISP. This permits you to redefine even the basic objects of INTERLISP. However, using this facility is not recommended as large parts of INTERLISP will either cease to work or work in strange ways. Moreover, the resulting code that you produce is not transportable to other systems.

Printing new datatypes may be difficult. However, using DEFPRINT (see Section 15.1.5), you may define one or more functions that print the structure of the datatype. You may also define how the objects are to be evaluated using the function DEFEVAL.

27.12.2 Fetching the Contents of an Object Field

Individual fields of an object may be fetched (or retrieved) using **FETCHFIELD**, which takes the form

```
Function:       FETCHFIELD
# Arguments:    2
Arguments:      1) a field descriptor, DESCRIPTOR
                2) an object, DATUM
Value:          The contents of the field.
```

DESCRIPTOR is a field descriptor returned that is returned by DECLAREDATATYPE. FETCHFIELD returns the contents of the field specified by DESCRIPTOR.

If the object is not an instance of the datatype for which DESCRIPTOR is a field descriptor, INTERLISP causes an error "DATUM OF INCORRECT TYPE".

27.12.3 Replacing the Contents of an Object Field

You may replace (or store into) the field of an object using **REPLACEFIELD**, which takes the form

```
Function:       REPLACEFIELD
# Arguments:    3
Arguments:      1) a field descriptor, DESCRIPTOR
                2) an object, DATUM
                3) a new value, NEWVALUE
Value:          The new value.
```

DESCRIPTOR is a field descriptor that is returned by DECLAREDATATYPE. REPLACEFIELD stores NEWVALUE as the contents of the specified field into the object.

If DATUM is not an instance of the object for which DESCRIPTOR is a field descriptor, INTERLISP causes an error "DATUM OF INCORRECT TYPE".

27.12.4 Creating an Instance of an Object

You may create an instance of an object using **NCREATE**, which takes the form

```
Function:       NCREATE
# Arguments:    2
Arguments:      1) an object name, TYPENAME
                2) a source object, FROM
Value:          The address of the new object.
```

NCREATE creates an instance of the specified object by allocating storage in memory for the instance.

If FROM is an instance of TYPENAME, the fields of the new object are initialized using the values of the corresponding fields in FROM.

NCREATE does not work for built-in datatypes such as ARRAYP, STRINGP, etc., because they have their own creation and initialization functions built into the INTERLISP virtual machine. INTERLISP will cause the error "ILLEGAL DATA TYPE" if TYPENAME is not a declared datatype.

27.12.5 Obtaining the Field Specifications

You may obtain the field specifications for a datatype using the function **GETFIELDSPECS**. It takes the form

```
Function:       GETFIELDSPECS
# Arguments:    1
Arguments:      1) an object name, TYPENAME
Value:          A list which is EQUAL to the FIELDS
                argument given to DECLAREDATATYPE.
```

GETFIELDSPECS returns a list of the field specifications for the specified object. If TYPENAME is not a currently declared object, it returns NIL. This allows you to dynamically declare objects by loading files containing the object declarations.

27.12.6 Obtaining the Field Descriptors

You may obtain the field descriptors of an object using **GETDESCRIPTORS**, which takes the form

```
Function:       GETDESCRIPTORS
# Arguments:    1
Arguments:      1) an object name, TYPENAME
Value:          A list of field descriptors.
```

GETDESCRIPTORS returns a list of field descriptors which is EQUAL to the value returned by DECLAREDATATYPE. Elements of this list may be used in FETCHFIELD and REPLACEFIELD.

If TYPENAME is not a currently declared datatype, GETDESCRIPTORS returns NIL.

27.12.7 Identifying User Datatypes

You may determine which datatypes have been defined by a user using **USERDATATYPES**, which takes the form

Function:	USERDATATYPES
# Arguments:	0
Arguments:	N/A
Value:	A list of the names of all currently declared user datatypes.

28

The History Package

The History Package is a set of commands and functions that allow you to refer to what you have done in the past. The primary motivation for the History Package is to provide a means whereby you may correct erroneous expressions and re-execute them. Each input that you type in is treated as an *event*. Associated with each event is a *history list* which maintains information relevant to the event. Entries on the history list consist of an input and its value. In addition, optional information such as side effects and formatting parameters may also be included on the history list.

INTERLISP maintains a list of events. As new events occur, existing events are aged. The "oldest" event is forgotten as each new event is entered onto the list. INTERLISP typically remembers about thirty events at one time although you may change this value via CHANGESLICE (see Section 28.6.6).

INTERLISP numbers events for future reference by printing an integer number when it is ready to receive the next input in the READ-EVAL-PRINT loop. These prompts run from 1 to 100 whence they roll over to 1 again. You may reference events by their numbers in history commands.

When the History Package is activated, each input that you type is placed on the history list after it is read, but before it is evaluated. Thus, if the command fails, the input expression is available for modification and eventual re-execution. In conjunction with the Programmer's Assistant (see Chapter 25), new functions and variables are noted and added to the spelling list. After a function or command is executed, its value is also saved on the history list for possible referral at a later time.

28.1 STRUCTURE OF THE HISTORY LIST

A *history list* has the following structure:

```
(alist event# size max-event#)
```

where:	alist	is a list of events with the most recent event first (i.e. at the front of the list)
	event#	is the event number of the most recent event
	size	is the maximum number of events to be retained
	max-event#	is the highest possible event number in multiples of 100.

INTERLISP maintains three history lists: LISPXHISTORY, ARCHIVELST, and EDITHISTORY. LISPXHISTORY records events associated with input typed to the READ-EVAL-PRINT loop while EDITHISTORY preserves your interactions with the INTERLISP Editor. Both are initialized to (NIL 0 30 100). ARCHIVELST is used to archive events when you type the command ARCHIVE.

You may disable the recording and maintaining of the history list for either the top-level interface or the Editor by setting LISPXHISTORY or EDITHISTORY to NIL, respectively.

Each history list has a maximum length—its time slice. As new events occur, they are placed at the front of the history list. That is, the history list is treated as a FIFO (first-in first-out) queue. The position of an event in the history list corresponds to its age. So, as new events are entered into the history list, older events recede into the past until they reach the threshold of the number of events to be remembered. When the numbers of events exceeds this threshold, the oldest events are forgotten. The amount of memory consumed in representing a history list can be enormous as demonstrated by a simple event below. You may selectively remember important events using the NAME and RETRIEVE commands (see Section 28.4.8).

Event Structure

An event, when it is recorded on a history list, has the form

(⟨input⟩ ⟨identifier⟩ ⟨value⟩ . ⟨properties⟩)

where:	⟨input⟩	is the expression typed in
	⟨identifier⟩	is the prompt character (which helps to identify the subsystem you are executing under)
	⟨value⟩	is the value of the expression after it has been evaluated
	⟨properties⟩	is a property list

History Package Properties

The property list of an event is used to store additional information about the event. The History Package supports the following properties:

SIDE	The side effects associated with the event.
ARCHIVE	Forces an event to be archived when it is about to be forgotten, i.e., when it is about to fall off the end of the history list.
GROUP	Used to support execution of a previous event.
HISTORY	Stores a history command that was applied to a previous event.
PRINT	Used for displaying the event, primarily by the Break Package and the ?? command.
USE-ARGS	Contains arguments for a history command.
...ARGS	Contains the expression for a history command.
ERROR	Contains information saved when an error occurs.
CONTEXT	Used in error processing.
LISPXPRINT	Stores information for calls to Programmer's Assistant functions.

Let us assume that I have just initialized an INTERLISP session and asked INTERLISP to load a file. The script that I would see at the terminal would appear as

```
Good Evening
2←(LOAD 'COMPLEX)
⟨KAISLER⟩COMPLEX..6
File Created 22-Nov-84 10:30
COMPLEXCOMS
3←(EDITV LISPXHISTORY)
```

To inspect the internals of LISPXHISTORY, you should apply the Editor to it to descend into the various levels. Be careful not to modify the history list or

you may not be able to undo erroneous events. We may apply Editor commands to navigate about the history list:

```
1*P
((& & &) 3 30 100)
1*1 P
((& _) (& _ ⟨KAISLER⟩COMPLEX..6 *LISPXPRINT* & SIDE &) (&
_ T SIDE & *LISPXPRINT* &))
2*1 P
((&) _)
2*1 P
((EDITV LISPXHISTORY))
2*0 0 2 P
((&) _ ⟨KAISLER⟩COMPLEX..6 *LISPXPRINT* (" " & "FILE
CREATED " "21-NOV-84 20:27:17" " " &) SIDE (-1 & & & & &
& & & & & & & & & & & --))
2*PP
(((LOAD (QUOTE COMPLEX))) _ ⟨KAISLER⟩COMPLEX..6
*LISPXPRINT*
(" " (PRINT ⟨KAISLER⟩COMPLEX..6 T) "FILE CREATED "
"21-NOV-84 20:27:17" " " (PRINT COMPLEXCOMS T T))
SIDE
(-1 (/SETTOPVAL CHANGEDFNSLST
                (FACTORIAL PRINT.ARRAY TRUNCATE
                RECIPROCATE
                          ROUNDED SIGN ROUNDTO FLOOR
                          PRINT.COMPLEX CMULT CZERO
                          CDIFFERENCE CPLUS IMAG REAL
                          COMPLEX))
    (/SETTOPVAL CHANGEDRECORDLST (COMPLEX))
    (/SETTOPVAL CHANGEDVARLST (COMPLEXVARS COMPLEXFNS
    COMPLEXCOMS))

    (/SETTOPVAL CHANGEDFNSLST
                (FACTORIAL PRINT.ARRAY TRUNCATE
                RECIPROCATE
                          ROUNDED SIGN ROUNDTO FLOOR
                          PRINT.COMPLEX CMULT CZERO
                          CDIFFERENCE CPLUS IMAG REAL
                          COMPLEX))
     (/SETTOPVAL CHANGEDRECORDLST (COMPLEX))
     (/SETTOPVAL CHANGEDVARLST (COMPLEXVARS COMPLEXFNS
     COMPLEXCOMS))
     (/SETTOPVAL FILELST NIL)
     (/RPLACD ((COMPLEXCOMS . T))
              NIL)
```

```
    (/RPLACA
        ((("21-Nov-84 20:27:17" . ⟨KAISLER⟩COMPLEX..6))
        FILEMAP
        ... ⟨rest of the filemap⟩ ...))
(/PUT-1 COMPLEX (FILE ((COMPLEXCOMS . T))))
(/SETTOPVAL LOADEDFILELST NIL)
(/PUT-1 COMPLEX (FILEMAP ⟨KAISLER⟩COMPLEX..6
        ... ⟨rest of the filemap⟩ ...))
... ⟨more⟩ ...
```

You can see that a considerable amount of information must be stored in order to be able to undo events at a later date.

28.2 UPDATING THE HISTORY LIST

The history list is updated through the auspices of LISPX (see Section 25.2) which functions as an EVAL-PRINT mechanism. When LISPX is given a value, it calls HISTORYSAVE to create a new event except for certain commands that are executed immediately. HISTORYSAVE (see Section 28.6.1) returns a template of the new event which is bound to LISPXHIST. When LISPX completes the evaluation of the input, it stores its value in LISPXHIST (as its CADDR). LISPXHIST also provides access to the properties associated with the current event.

28.3 EVENT SPECIFICATION

Events are associated with user inputs. The key to successful use of the History Package is to describe the event which you want to reference.

An *event specification* is a statement that describes a sequence of one or more events for a history command. A specification is drawn from the following set of phrases:

`FROM ⟨1⟩ THRU ⟨2⟩`	Specifies the sequence of events beginning at address ⟨1⟩ up to and including address ⟨2⟩. Address ⟨1⟩ may be more recent than address ⟨2⟩ thus providing for the processing of events in reverse order.
`FROM ⟨1⟩ TO ⟨2⟩`	Specifies the sequence of events beginning at address ⟨1⟩ up to but not including address ⟨2⟩. Reverse order is allowed.
`FROM ⟨1⟩`	Equivalent to FROM ⟨1⟩ TO -1.

THRU ⟨2⟩	Equivalent to FROM -1 THRU ⟨2⟩.
TO ⟨2⟩	Equivalent to FROM -1 TO ⟨2⟩.
⟨1⟩ AND ⟨2⟩ AND ...	Specifies a sequence of event specifications.
ALL ⟨1⟩	Specifies all events satisfying the form given as ⟨1⟩. For example, ALL LOAD would consider all events having LOAD as the function.
⟨empty⟩	No specification is the same as -1, i.e., the last event. However, if the last event was an UNDO, then it corresponds to a -2.
@ ⟨literal-atom⟩	If ⟨literal-atom⟩ is the name of a command defined via NAME (see Section 28.4.8), this form specifies the event that defined ⟨literal-atom⟩, i.e., the event that saved the event which is the value of ⟨literal-atom⟩.
@@ ⟨1⟩	An event specification that is interpreted with respect to the archived history list.

Note that ⟨1⟩ and ⟨2⟩ may refer to the index of an event or the name of the function corresponding to the event.

As an example, FROM TCOMPL could specify a sequence of events beginning with a recent compilation and progressing up to the most recent event.

If the History Package cannot find any events that match the specification, it attempts spelling correction on the words in the specification using LISPX-FINDSPLST to correct possible misspellings. If the event specification still fails to identify any events, an error is generated.

28.3.1 Event Addresses

An *event address*, such as ⟨1⟩ above, is a specification of an event number in the history list. An event is located by moving an imaginary cursor up or down the history list. The event selected is the one that lies "under" the cursor when there are no more commands to be interpreted in the event specification. Each command always moves the cursor. Addresses may be specified in the following ways:

⟨n⟩	Move forward ⟨n⟩ events in the history list where n => 1.
←n⟩	Move backward -n events where n => 1.
← ⟨atom⟩	Specifies an event whose function matches ⟨atom⟩. This form is used to select an event where ⟨atom⟩ was the function used in the event.
←	Indicates the next search is to go forward rather than backwards; if given as the first address indicates that the search begins at the last element of the list.
F	Indicates that the next object is a literal that is to be searched for, e.g., F -4 searches for an event containing a -4 rather than moving backwards four events in the history list.
=	Indicates that the next object in the address is to be matched against values of events rather than the inputs.
\	Indicates the last event located.
SUCHTHAT ⟨pred⟩	Specifies an event for which ⟨pred⟩, a predicate of two arguments, is true. The first argument is the input portion of the event. The second argument is the event itself. For example, SUCHTHAT (LAMBDA (X Y) (MEMBER '*ERROR* Y)) specifies an event in which an error occurred.

⟨atom⟩	Specifies the name of a command defined by the NAME command (see Section 28.4.9).
⟨pattern⟩	Anything else which may be matched against the input of an event.

28.4 HISTORY COMMANDS

The History Package provides a powerful, flexible set of commands for modifying and re-executing expressions in the history list. All history commands take an event specification which is signified by ⟨event⟩. If ⟨event⟩ is omitted from a history command, then the value -1 is assumed (i.e., apply it to the previous event).

28.4.1 Re-executing Previous Expressions

You may re-execute a previous expression on the history list by executing the **REDO** command. The simplest form of this command is

```
REDO ⟨event⟩
```

which executes the event(s) specified.

```
REDO FROM -6 TO -3
```

will do three events beginning six deep in the history list. Another form that is useful is

```
REDO ← ⟨atom⟩
```

which looks for an entry in the history list whose function corresponds to ⟨atom⟩ and re-executes it. You may have performed a compilation, noted an error, fixed it via the Editor, and now wish to redo the compilation. You might give the command REDO COMPILE.

You may re-execute an event a number of times using the form

```
REDO ⟨event⟩ ⟨n⟩ TIMES
```

which re-executes the specified event ⟨n⟩ times where ⟨n⟩ => 1. Given a list of M (M => 5) elements, if the last event were

```
(SETQ alist (CDR alist))
```

then

```
REDO 5 TIMES
```

would strip off the first five elements of the list.

You may re-execute an event based on the value of some S-expression which is evaluated prior to interpreting the history command. The formats are

```
REDO ⟨event⟩ WHILE ⟨form⟩
REDO ⟨event⟩ UNTIL ⟨form⟩
```

Suppose the last two events consisted of the following sequence:

```
(SETQ myfilelist (CDR myfilelist))
(MAKEFILE (CAR myfilelist))
```

which runs through a list of files that I have maintained and creates a new version of each according to its File Package commands. I could execute these two expressions until MYFILELIST is exhausted by issuing the followng history command:

```
REDO FROM -2 UNTIL (NULL MYFILELIST)
```

which terminates when MYFILELIST is exhausted.

The REPEAT Command

Alternative forms of these two commands are

```
REPEAT ⟨event⟩
REPEAT ⟨event⟩ WHILE/UNTIL ⟨form⟩
```

The former is equivalent to REDO ⟨event⟩ WHILE T and continues to execute until an error occurs or the user interrupts execution via keyboard control.

The general form of REDO (and its alternative REPEAT) allow you to perform multiple executions of a previous event. A system variable REDOCNT is initialized to 0 when the command is interpreted. On each iteration, REDOCNT is incremented by 1. If the command terminates properly, the numer of iterations will be displayed. You may access REDOCNT in ⟨form⟩.

28.4.2 Argument Substitution

You may substitute a different expression in the input of an event and re-execute that event. Let the previous event be

```
(SETQ sin.of.point (SIN (DEGREES.TO.RADIANS point)))
```

You could also calculate the COS.OF.POINT by issuing the following history command:

```
USE COS.OF.POINT FOR SIN.OF.POINT AND COS FOR SIN IN -1
```

which performs the appropriate substitutions and re-executes the expression.

The general form of the USE command is

```
USE ⟨expression[1]⟩ FOR ⟨argument[1]⟩ AND ...
    ⟨expression[n]⟩ FOR ⟨argument[n]⟩ IN ⟨event⟩
```

The USE command requires three pieces of information:

1. The expressions to be substituted,
2. The arguments to be substituted for,
3. An event specification.

An expression may be preceded by a !. This indicates that it is to be substituted as a segment, i.e., it replaces only a portion of the input statement. Assume that ANGLES-IN-RADIANS is a variable. To substitute this variable into the expression, you could use

```
USE ! ANGLES-IN-RADIANS FOR (DEGREES-TO-RADIANS POINT)
```

You may omit the arguments in a USE command. There are two cases to be considered:

1. The ⟨event⟩ is a USE command. The substitution takes place for the same arguments as the previous USE command.

   We could also compute the tangent of the point by issuing the command

   ```
   USE ! TAN IN -1
   ```

2. The ⟨event⟩ is not a USE command. The substitution is performed for the operator in the specified event:

   a. If it was LISPX input, then the expression is substituted for the function in the input expression.

      ```
      USE CAR IN CDR
      ```

   b. If it was an edit command, then the expression is substituted for the name of the command.

You may also omit the IN phrase in the USE command while specifying the arguments. In this case, the History Package uses the first item in arguments to build an event specification. It does so by implicitly inserting an F command with the first argument into an IN phrase in the USE command.

So far, we have looked at USE commands that specify a one-for-one substitution between the expressions and the arguments. The History Package can distribute several expressions over the same argument. Various combinations are available.

```
USE BALTIMORE MARYLAND FOR RICHMOND VIRGINIA
```

is equivalent to saying

```
USE BALTIMORE FOR RICHMOND AND MARYLAND FOR VIRGINIA
```

We might also say

```
USE BALTIMORE ELKTON HAGERSTOWN FOR CITY
```

which substitutes BALTIMORE for CITY and executes the event, then substitutes ELKTON for CITY, and executes the event, and finally substitutes HAGERSTOWN for CITY.

An alternative form of the USE command, ⟨ESC⟩ (echoed as $), is used to specify character substitutions in strings and literal atoms. It basic forms are

```
$ ⟨x⟩ FOR ⟨y⟩ IN ⟨event⟩ or
$ ⟨y⟩ ⟨x⟩ IN ⟨event⟩
$ ⟨y⟩ = ⟨x⟩ IN ⟨event⟩
$ ⟨y⟩ -⟩ ⟨x⟩ IN ⟨event⟩
```

which are equivalent to

```
USE $⟨x⟩$ FOR $⟨y⟩$ IN ⟨event⟩
```

as used in

```
$ COS FOR SIN IN -1
```

would have the same effect as the previous history command given above. The event would be re-executed as

```
(SETQ cos.of.point (cos (DEGREES.TO.RADIANS point)))
```

$ may only be used to specify one substitution at a time. After it finds an event, $ determines if an error was involved in that event. If the indicated character substitution can be performed in the offending object, the offender, $ performs the substitution in the offender only and then substitutes the result for the original offender throughout the remainder of the event. This is the most common use of the $ command—to correct a spelling or other trivial character-based error in a command typed at the top level. Consider the following example:

```
←(LOAD 'COOMPLEX)
FILE NOT FOUND
COOMPLEX

←$ OO FOR O in LOAD
OO->O
<KAISLER>COMPLEX..6
FILE CREATED 21-Nov-84 20:27:17
COMPLEXCOMS
```

$ never searches for an error. It performs the replacement in the offender only if the specified event caused an error.

28.4.3 Editing a Previous Event

Sometimes the previous expression is so complex that it becomes rather tedious to "write" the history commands to correct it. In these cases, it is easier to edit the expression and then re-execute it. You can edit a previous event by issuing the **FIX** command. It takes the form

```
FIX <event>
```

FIX puts you into the INTERLISP Editor with a copy of the input expression which has been extracted from the event entry on the history list. When you exit the Editor with an OK, the result is substituted into the event and re-executed. Suppose we are defining a function:

```
←(DEFINEQ
    (ADD3 (X Y Z)
          (IPLUS X YZ)))
(ADD3)
```

In my haste to enter the function definition, I concatenated the Y and Z. Instead of retyping the definition, I can fix it as follows:

```
←FIX
edit
```

```
*P
(DEFINEQ (ADD3 & & &))
*2 4 P
(IPLUS X YZ)
*(R YZ Y)
*(N Z)
*OK
(ADD3 redefined)
(ADD3)
```

If the edition is relatively simple, you can specify a sequence of Editor commands following the FIX command which are applied to the expression. After the last Editor command, an implicit OK is assumed which forces the result to be re-executed. Thus, we could have performed the same editing as follows:

```
←(DEFINEQ
    (ADD3 (X Y Z)
        (IPLUS X YZ)))
(ADD3)
←FIX defineq-2 4 P (R YZ Y)
...(N Z)
...
←
```

whence the Editor does not type *edit* or wait for an OK after executing the commands.

28.4.4 Retrying an Event

A particularly difficult problem to assess is the expression which enters an infinite loop or takes an unusually long time to execute. In these cases, you want to enter the Break Package after a given period of time to determine what has happened. The **RETRY** command allows you to force breaks if a previous event executes in such a way as to cause errors that are not trapped. Its format is

```
RETRY ⟨event⟩
```

If I wanted to reset my erroneous definition of ADD3 in order to demonstrate an alternative method of correcting it, I could retry the original event as follows:

```
←RETRY 17
(ADD3 redefined)
(ADD3)
```

28.4.5 Printing the History List

In manipulating the history list, you may forget what events have previously occurred (particularly if your terminal scrolls off the top as mine does—so does INTERLISP-D's typescript window). You can inspect the history list by issuing the **??** command. ?? prints the history list. Unlike other commands, if the ⟨event⟩ is omitted, ?? will display the entire history list for you (beginning with the most recent event).

Note that ?? commands are never entered onto history lists. Thus, they do not affect the relative event numbers. For example, ?? -1 refers to the event immediately preceding the ?? command. However, ?? does print the history command associated with an event. You may distinguish the history commands by noting that they are not preceded by a prompt character.

?? is implemented by the function PRINTHISTORY which you may also invoke directly. Consider the following example:

```
←??
37. RETRY 17
(DEFINEQ (ADD3 (X Y Z) (* edited "24-Nov-84 09:17")
(IPLUS X Y Z)))
(ADD3 redefined)
(ADD3)
36. RETRY 2
2 ?            "because event 2 has fallen off the end of
                the history list"
35. FIX DEFINEQ-2 4 P (R YZ Y) (N Z)
34←
```

For each event, ?? prints the event number, the prompt character, the input line(s) and the value(s). If the event was a Programmer's Assistant command that "unread" other input lines, the Programmer's Assistant command will be printed without the prompt character to show that they are not stored as input, and the input lines will be printed with prompt characters.

28.4.6 Undoing the Effects of Events

You may undo the effects of a previous expression or command by executing the **UNDO** command. Its format is

```
UNDO ⟨event⟩
```

UNDO undoes the side effects of the specified events. For each event that is undone, the History Package prints a message of the form

```
⟨function⟩ UNDONE
```

where ⟨function⟩ is in the input of the event. If nothing is undone because nothing was saved, the History Package merely types NOTHING SAVED.

```
←(DEFINEQ (GET.HEAD (LST) (CAR LST)))
(GET.HEAD)

←UNDO DEFINEQ
DEFINEQ undone

←(PP GET.HEAD)
(GET.HEAD not printable)
(GET.HEAD)
```

If nothing was undone because the event's side effects were already undone, the History Package prints the message ALREADY UNDONE. When no event is specified, UNDO searches back through the history list until it finds an event that has side effects which were not undone and where the input is not another UNDO command. It undoes that event.

You may undo the effects of UNDO commands.

```
←UNDO UNDO
UNDO undone

←(PP GET.HEAD)
(GET.HEAD
     (LAMBDA (LST) **COMMENT**
          (CAR LST)))
```

In general, undoing events in the reverse order of execution will restore all INTERLISP pointers to their correct previous values. The easiest way to say this is UNDO THRU -5. If you undo events in any other order, the state of the system is unpredictable because the expressions may have been dependent upon the sequence of execution.

An alternative version of the UNDO command has the following form:

```
UNDO ⟨event⟩ : ⟨pattern1⟩ ... ⟨patternN⟩
```

Each ⟨pattern[i]⟩ is matched to a message printed by DWIM in the event specified by ⟨event⟩. The side effects of the corresponding DWIM corrections are undone. If DWIM had printed the following message:

```
LOOAD [IN GET.FILES] -⟩ LOAD
```

performing the command

```
UNDO : LOAD
```

would undo the correction.

Note that some of the DWIM messages include strings. Merely typing an atomic form would not find the string. You must surround it by ⟨ESC⟩ characters to locate it.

28.4.7 Correcting Errors via DWIM

When an error occurs, you can attempt to have DWIM fix it with information it already possesses such as spelling lists, etc. You do so by executing the history command DWIM which invokes DWIM to correct and re-execute the expression. Suppose that you had typed the expression

```
(GETPROP 'state 'capital)
```

and had received NIL as the answer. Of course, the property name should be CAPITOL. If you had previously executed

```
(PUTPROP 'state 'capitol)
```

DWIM would have enough knowledge to correct the spelling of the property and retrieve the value, if any.

Associated with certain commonly used functions are heuristics about how they are used. When a function fails, DWIM determines if one of these functions was involved in the input. If so, it uses the heuristic information to correct the erroneous condition. Usually, this heuristic information concerns spelling and the nature of the arguments. DWIM knows about the following functions:

1. GETD, FNTYP, and MOVD all take as their first argument an atom that should have a definition.
2. GETP, GETPROP, GETPROPLIST, PUT, and PUTPROP take as a first argument an atom that should have a non-NIL property list and a second argument which should be a property on that property list or a member of SYSPROPS.
3. SET, SETQ, and SETQQ take as a second argument a variable that is bound to some value which is assigned as the value of the first argument.
4. LOADFNS and EDITF take as an argument the name of a function that is contained on one of the files in FILELST (i.e., the function has already been loaded once).

If any of these conditions are not true for the expression, DWIM attempts to make them true through spelling correction. If the condition is made true, the corrected event is re-executed.

If an error occurred in the indicated event due to a misspelling, DWIM can look for an atom that is "close to" the word that was originally intended. To do so, it appends $$ (i.e., two escapes ⟨esc⟩) onto the end of the offender, substi-

tutes it into the event, and tries again. The History Package will then search for an event that has a word close to the offender and attempt to re-execute that event.

```
(MOVE 3 2 TO AFTER CONDD 1)
```

will cause the Editor to respond CONDD?. When you type DWIM, it substitutes CONDD⟨esc⟩⟨esc⟩ into the event, which causes

```
(MOVE 3 2 TO AFTER CONDD⟨esc⟩⟨esc⟩ 1)
```

to be executed. It searches for an atom close to CONDD, which is COND, and this event is re-executed.

If these procedures fail, DWIM displays the message

```
Unable to figure out what you meant in: ⟨event⟩
```

28.4.8 Saving and Retrieving Events

You may save and restore events on the property list of an atom. The form of these commands is

```
NAME ⟨literal-atom⟩ ⟨event⟩
RETRIEVE ⟨atom⟩
```

The specified events, including the side effects, are stored on the property list of the atom under the property HISTORY. Events saved on the property list of the ⟨literal-atom⟩ may be retrieved using the event specification form: @⟨literal-atom⟩.

This feature is quite useful when you are executing a sequence of expressions associated with an atom which you might want to repeat several times throughout the session. In particular, you may make beneficial use of it during the debugging of a program when you enter expressions to look at the values of various variables used by the program.

In effect, you are describing a new command for the History Package by associating events with an atom. You may execute the command as a history command by merely typing its name.

```
24←(DEFINEQ
          (CLEAR.ARRAY (XARRAY)
               (FOR I FROM 1 TO (ARRAYSIZE XARRAY)
                    DO (SETA XARRAY I 0))))
(CLEAR.ARRAY)
25←(CLEAR.ARRAY XARRAY)
NIL
```

```
26←NAME CLEARA 25
CLEARA

27←CLEARA
NIL
```

which is equivalent to REDO 25. However, if ⟨literal-atom⟩ has a top-level value, then the value of the atom is returned (i.e., the atom is evaluated). Thus, you must be careful to define new history commands as atoms that do not have top-level values.

Commands defined by NAME may be parametrized. The format of the NAME command is

```
NAME ⟨literal-atom⟩ (⟨arguments⟩) : ⟨event⟩
```

or

```
NAME ⟨literal-atom⟩ ... ⟨arguments⟩ ... : ⟨event⟩
```

where the ⟨arguments⟩ are interpreted as if a USE command had been given. Both !s and $s may also be used in specifying the structure of the arguments. Consider the following example:

```
22←(PUTD 'X (COPY (GETPROP 'Y 'EXPR)))
(X)

23←NAME MOVE.DEFINITION X Y : PUTD
MOVE.DEFINITION
```

which defines the MOVE.DEFINITION command. Then, typing the statement

```
24←MOVE DUMP.MATRIX DUMP.ARRAY
```

would result in the execution of the following expression:

```
(PUTD 'DUMP.MATRIX (COPY (GETPROP 'DUMP.ARRAY 'EXPR)))
```

which is equivalent to executing

```
USE DUMP.MATRIX DUMP.ARRAY FOR X Y IN MOVE.DEFINITION
```

Commands, defined by NAME without arguments, which are invoked with arguments will cause errors.

RETRIEVE extracts the events from the atom and re-enters them on the history list. It does not re-execute them. If ⟨literal-atom⟩ did not previously appear in a NAME command, the History Package generates an error.

```
RETRIEVE ⟨example⟩
```

Once you have retrieved events from the HISTORY property of an atom, you may apply any history command to those events (including saving them on another atom).

You may execute the events stored under the HISTORY property of an atom by executing a REDO command:

```
REDO ⟨literal-atom⟩
```

is equivalent to

```
RETRIEVE ⟨literal-atom⟩
REDO
```

You may also see what events have been stored under the HISTORY property of an atom by executing

```
?? ⟨literal-atom⟩
```

Suppose you have defined a new command via the NAME command. This command records events which have been applied to one or more variables. You may undo the effects of the events associated with the command by executing the history command BEFORE, which takes the form

```
BEFORE ⟨literal-atom⟩
```

You may re-execute the events by executing the history command AFTER, which takes the form

```
AFTER ⟨literal-atom⟩
```

BEFORE and AFTER allow you to "flip" back and forth between two program states in order to test the effect of modifications to data structures or variable values.

Both BEFORE and AFTER will generate an error if their argument was not previously defined via a NAME command.

28.4.9 Archiving Events

The History Package maintains a third history list which is a permanent history list. You may archive events on that history list by executing the ARCHIVE command which takes the form

```
ARCHIVE ⟨event⟩
```

Events recorded on the archival history list may be referenced by preceding the event specification by @@. The @@ specification may be used with any history command subject to other restrictions as previously noted.

Events may automatically be archived when they are about to be forgotten by defining ARCHIVEFN (see Section 28.5.3) to do the proper thing.

28.4.10 Forgetting Side Effects

As we noted previously, events may (and often do) have side effects. You may forget the side effects associated with an event by executing the history command **FORGET**, which takes the form

```
FORGET ⟨event⟩
```

If ⟨event⟩ is omitted, the side effects for the entire history list are forgotten. Note that FORGETing is not undoable.

Many INTERLISP functions preserve old pointers in the side effects of an event when new values are created. Among these functions are SETx, PUTD, REMPROPS, COPY, etc. The old pointers are not garbage collected until the event is forgotten from the history list. You may force garbage collection (at the next appropriate time) of these pointers by FORGETting them from the side effects of their respective events.

The IRM notes that this function is provided for users with space problems. You probably do not need to worry about this situation if you are running under INTERLISP-D unless you have reset the maximum number of events to a fairly large number and are running a problem that has very large and complex data structures.

28.4.11 Remembering Events

You may remember events across sessions by saving the events in a file. The **REMEMBER** command causes the File Package to remember events as instances of the File Package type EXPRESSIONS? whenever a FILES?, MAKEFILES, or CLEANUP (see Section 17.3.6) is executed. The File Package writes a command of the form (P ⟨event⟩) into the COMS of the specified file.

28.4.12 Printing Property Lists

You may print the property list of an atom by executing the history command **PL**, which takes the form

```
PL ⟨atom⟩
```

PL prints the property list of the specified atom with the print level (see Section 15.3) temporarily reset to (2 . 3).

```
←PL MASTERSCOPE
FILEDATES : (("  4-Aug-83 23:47:11" .
〈NEWLISP〉MASTERSCOPE.;1))
```

PL is equivalent to executing GETPROPLIST upon the name of the atom.

28.4.13 Printing Atom Bindings

You may print the binding (i.e., the value) of an atom by executing the history command **PB**, which takes the form

```
PB 〈atom〉
```

Its primary purpose is to allow you to inspect the values of atoms which may have complex data structures.

```
←PB LISPXHISTORY
@ TOP : ((& & --) 17 30 --)
```

Note that the effect of PB is mediated by the current setting of the print level.

```
←PB X
@ TOP : NOBIND
```

Note that if the atom is not bound, the History Package will not attempt spelling correction or generate an error.

28.4.14 Analyzing Errors

When an error occurs in an expression, you can ask the Programmer's Assistant to attempt an analysis of the error. You do so by executing the history command:

```
?
```

The Programmer's Assistant attempts to analyze the nature and cause of the error using context information saved when the input of the event was executed. Based on this analysis, it will try to put you into the Editor at the proper location to correct the problem. Consider the following example:

```
←(DEFINEQ (GET.HEAD (LST) (CARR LST)))
(GET.HEAD)
←(GET.HEAD '(MUSIAL MANTLE MARIS))
UNDEFINED CAR OF FORM
CARR [in GET.HEAD] in [CARR LST]
```

```
←?
edit
(LAMBDA (LST) **COMMENT** (CARR LST))
1*
```

28.4.15 Bypassing the Programmer's Assistant

Every line that you type into LISPX will be examined and processed by the Programmer's Assistant. You may force this processing to be bypassed by entering a command of the form

```
: ⟨expression⟩
```

The : allows you to enter input to the system without having it processed by the Programmer's Assistant.

28.4.16 Preventing History List Recording

Every expression submitted to the top-level READ-EVAL-PRINT loop is recorded on the history list unless it is one of the excepted Programmer's Assistant commands. Note that the amount of information saved for each event varies with the nature of the expression that you type in (see the example above). You may prevent the recording of an expression on the history list using the **SHH** command, which takes the form

```
SHH ⟨statement⟩
```

where ⟨statement⟩ may be either an S-expression or a Programmer's Assistant command.

28.5 HISTORY PACKAGE VARIABLES

The History Package uses a number of variables to determine how it will respond to certain commands and how it will treat certain events. These variables are described in this section.

28.5.1 LISPX History Macros

LISPXHISTORYMACROS is a variable whose value is a list of elements of the form

```
(⟨command⟩ ⟨definition⟩)
```

LISPXHISTORYMACROS allows you to define your own History Package commands. When ⟨command⟩ is recognized on type-in, ⟨definition⟩ is evalu-

ated. However, unlike LISPXMACROS, its result is treated as a set of expressions which are to be unread. That is, the result of evaluating ⟨definition⟩ is placed at the front of READBUFFER (see LISPXUNREAD). It is permissible to return NIL as the result of evaluating ⟨definition⟩ which means nothing else is to be done.

LISPXHISTORYMACROS takes as its initial value (comments added for understanding in this listing)

```
((QU
     (*
          Quit from INTERLISP
     )
     NIL
     (LOGOUT)
 (BUF
     (*
          Put input into the read buffer. TTYIN is an
          INTERLISP-D function.
     )
      NIL
      (TTYIN LISPXID
             NIL
             NIL
             'EVALQT
             NIL
             NIL
             (LIST
                 (COND
                    (LISPXLINE
                         (VALUEOF LISPXLINE))
                    (T
                         (CADDR (CAAR LISPXHISTORY)))))))
 (DA
     (PROGN
          (*
               Print the date and time on the terminal;
               and move to the next line.
          )
          (LISXPRIN1 (DATE) T)
          (TERPRI T)))
 (; NIL NIL)
 (PB
     (*
          A macro definition for the PB command.
     )
```

```
        (MAPC LISPXLINE
              (FUNCTION
                  (LAMBDA (X)
                       (PRINTBINDINGS X
                                      (AND
                                           (EQ LISPXID ':)
                                           LASTPOS))))))
(PL
    (*
         A macro definition for the PL command.
    )
    (COND
         (LISPXLINE
              (PRINTPROPS (CAR LISPXLINE)))
         (T
              '(E PL))))
(??
    NIL
    (PROG (TEMP)
          (RESETVARS
             ((PRETTYTRANFLG T))
             (RESETFORM
                (OUTPUT T)
                (PRINTDEF
                   (COND
                      ((NULL
                          (CDAR
                             (SETQ TEMP
                                   (LISPXFIND
                                   LISPXHISTORY
                                              LISPXLINE
                                              'ENTRY))))
                          (CAAR TEMP))
                      (T
                          (CAR TEMP)))
                   NIL
                   T)))
          (TERPRI T)
          (RETURN NIL)))
(TYPE-AHEAD
    (LISPXTYPEAHEAD)))
```

28.5.2 History Package Forms

Several variables have as their values a list of expressions to be evaluated when certain functions are executed:

PROMPTCHARFORMS	A list of expressions to be executed each time PROMPTCHAR (see Section 25.6) is called.
HISTORYSAVEFORMS	A list of expressions to be executed each time HISTORYSAVE (see Section 28.6.1) is called.

Note that expressions on PROMPTCHARFORMS will be evaluated before the prompt character is printed. The expressions on HISTORYSAVEFORMS are executed after the user has completed type-in, but before the input is evaluated. Thus, you may effectively encapsulate the user's interaction with the system (or subsystem).

RESETFORMS	A list of expressions to be evaluated whenever (RESET) is executed, when the user types CTRL-D, or when the user types CTRL-C followed by START. See Section 25.7 for more information.

28.5.3 The Archival Function

ARCHIVEFN is a History Package variable whose value determines whether or not an event about to be forgotten will be archived or not. The value of ARCHIVEFN is initially NIL and undefined so that all events passing the age threshold will be forgotten. You may set ARCHIVEFN to T whence the definition stored in the function definition cell of ARCHIVEFN will be called to determine how to process the event.

The definition stored in the function cell of ARCHIVEFN must be a LAMBDA expression of two arguments: the input portion of the event and the entire event. If the function returns T, the event is archived on ARCHIVELST. ARCHIVELST is initially NIL.

To record all events concerning the loading of files, you would set ARCHIVEFN to T and define it as follows:

```
(LAMBDA (INPUT EVENT)
     (EQUAL (CAR INPUT) 'LOAD))
```

ARCHIVEFLG is a flag that determines how events referenced by history commands are to be processed. The rationale is that if you reference a previous event by a history command, it must be important (i.e., may be referenced again). Thus, when it falls off the end of the history list, it should be archived. If ARCHIVEFLG is non-NIL, the History Package automatically marks every event referenced by a history command so that it will be archived. AR-

CHIVEFLG is initially T. Events are marked for archiving by placing the property *ARCHIVE* with value T in the event entry.

ARCHIVELST is initialized to (NIL 0 50 100) so that it does not have an indefinite length.

28.5.4 The Value of an Event

The History Package variable IT always stores the value of the most recent event. It may be used in succeeding expressions or history commands just like any other variable. Consider the following example:

```
←(SQRT (TIMES 100 50))
70.71068

←IT
70.71068

←(SQRT IT)
8.408964
```

28.6 HISTORY PACKAGE FUNCTIONS

The History Package provides a number of functions that allow you programmatic access to the history lists from within your program or that allow you to maintain you own history lists that are application dependent. The latter feature is very useful in many artificial intelligence programs that require a history of decisions in order to implement an explanation facility. This section will discuss the History Package functions.

28.6.1 Recording a History Event

HISTORYSAVE records an event on a history list. It takes the form

```
Function:       HISTORYSAVE

# Arguments:    6

Arguments:      1) a history list, HISTORY
                2) an event identifier, ID
                3) an input specification, INPUT1
                4) an input specification, INPUT2
                5) an input specification, INPUT3
                6) a list of properties, PROPS

Value:          The new event identifier.
```

HISTORYSAVE creates a new event with an event identifier having the value of ID. The value field of the event is initialized to bell (?). The event has the

associated properties and values that are the value of PROPS. If the size of HISTORY has reached its maximum, the last event (e.g., the oldest) is removed from the list.

INPUT1, INPUT2, and INPUT3 describe the possible forms of the input sequence that produced the event:

1. If INPUT1 is non-NIL, the input entry is of the form

```
(INPUT1 INPUT2 . INPUT3)
```

2. If INPUT1 is NIL and INPUT2 is non-NIL, the input entry is of the form

```
(INPUT2 . INPUT3)
```

3. Otherwise, the input entry is just INPUT3.

The value of HISTORYSAVE is the new event.

If REREADFLG in non-NIL, and the most recent event records a command that produced this input, HISTORYSAVE does not create a new event but merely updates that event's *GROUP* property with an entry of the form (⟨input⟩ ⟨id⟩ ⟨bell⟩ . ⟨props⟩).

```
←SHH (HISTORYSAVE LISPXHISTORY 54 'HISTORYSAVE)
((HISTORYSAVE NIL) 54)
```

which is the structure of the new event. Note that you may evaluate a value and place it in the event using RPLACD.

HISTORYSAVE uses HISTORYSAVEFORMS when it creates a new event. If HISTORYSAVEFORMS is non-NIL, it should be a list of expressions that is evaluated (under ERRORSET protection) each time a new event is created. These expressions may use the values of the arguments to HISTORYSAVE as well as the variable EVENT which is the value that HISTORYSAVE is about to return.

28.6.2 Locating a History Event

HISTORYFIND searches the history list to find a specified event. It returns the tail of the history list beginning with the event that was looked for. It takes the form

```
Function:       HISTORYFIND
# Arguments:    4
Arguments:      1) a history list, HISTORY
                2) an index into the history list, INDEX
```

```
                3) the maximum event #, MOD
                4) an event specification, EVENTADR
Value:          The tail of the history list beginning
                with the specified event; otherwise, an
                error.
```

Consider the following example:

```
2←(SETQ COMPOSERS
        (LIST 'BACH 'BEETHOVEN 'COPELAND 'MOZART))
(BACH BEETHOVEN COPELAND MOZART)

3←PB COMPOSERS
@ TOP : (BACH BEETHOVEN COPELAND MOZART)

4←(HISTORYFIND (CAR LISPXHISTORY) 2 4)
((NIL _ *HISTORY* ((PB COMPOSERS)) *GROUP* NIL) (((SETQ
COMPOSERS (LIST (QUOTE BACH) (QUOTE BEETHOVEN) (QUOTE
COPELAND) (QUOTE MOZART)))) _ (BACH BEETHOVEN COPELAND
MOZART) SIDE (4 (UNDOSET NIL COMPOSERS NOBIND) (ADDSPELL2
COMPOSERS ("spellseparator" ... ⟨rest of spelling list⟩
...
```

Note that the first argument to HISTORYFIND uses the CAR of the actual history list because it should be the history list itself.

28.6.3 Locating Events by Specification

LISPXFIND searches a history list and returns a structure consisting of the events that match a given specification. It takes the form

```
Function:       LISPXFIND
# Arguments:    4
Arguments:      1) a history list, HISTORY
                2) an event specification, EV
                3) a format specification, TYPE
                4) a backup flag, BACKUP
Value:          As described below.
```

LISPXFIND attempts to locate the corresponding history events to the specification given by EV. EV is an event specification. TYPE specifies the format of the value to be returned by LISPXFIND. It is chosen from ENTRY, ENTRIES, COPY, COPIES, INPUT, or REDO. LISPXFIND parses EV and call HISTORYFIND to locate and assemble the specified events.

```
2←(SETQ COMPOSERS
        (LIST 'BACH 'BEETHOVEN 'COPELAND 'MOZART))
(BACH BEETHOVEN COPELAND MOZART)

3←PB COMPOSERS
@ TOP : (BACH BEETHOVEN COPELAND MOZART)

←(LISPXFIND LISPXHISTORY '(SETQ) 'ENTRY T)
(((SETQ COMPOSERS (LIST (QUOTE BACH) (QUOTE BEETHOVEN)
(QUOTE COPELAND) (QUOTE MOZART)))) _ (BACH BEETHOVEN
COPELAND MOZART) SIDE (4 UNDOSET NIL COMPOSERS NOBIND)
(ADDSPELL2 ...)))
```

BACKUP is a flag that is interpreted as follows:

1. If BACKUP is T, LISPXFIND looks for an event in the history list prior to the insertion of the current event so that it will not refer to itself.
2. If EV is NIL and the last event was the Programmer's Assistant command UNDO, the next to last event is retrieved from the history list. This permits you to type UNDO followed by REDO or USE.
3. ARCHIVELST will be substituted in place of the specified history list if EV contains @@ within its value.
4. If EV contains @, LISPXFIND will retrieve the event from the corresponding literal atom rather than the specified history list.

I recommend that you always use LISPXFIND with BACKUP set to T.

28.6.4 Extracting a History Event

ENTRY# extracts an event from a history list. It takes the form

```
Function:       ENTRY#

# Arguments:    2

Argument:       1) a history list, HISTORY
                2) an event specification, EVENT

Value:          The event number for EVENT.
```

```
←(ENTRY# LISPXHISTORY '(SETQ))
100
```

28.6.5 Obtaining an Event's Value

VALUEOF obtains the value of an event on the current history list. It takes the form

```
Function:       VALUEOF
# Arguments:    1
Argument:       1) an event specification, EVENT
Value:          The value of the corresponding event.
```

VALUEOF is an NLAMBDA, nospread function. Consider the following example:

```
←(VALUEOF 11)
(BACH BEETHOVEN COPELAND MOZART)
```

Note that if VALUEOF cannot find a value for the specified event, it returns the maximum event number.

28.6.6 Changing a History List's Timeslice

CHANGESLICE allows you to change the timeslice for a history list. As you remember, the timeslice is the number of events that will be recorded on the history list. It takes the form

```
Function:       CHANGESLICE
# Arguments:    2
Arguments:      1) a timeslice, N
                2) a history list, HISTORY
Value:          The value of timeslice.
```

The effect of increasing the time slice is a gradual lengthening of the history list as new events are recorded. Decreasing the time slice will cause the excising of enough events to reduce the history list to the proper size. Because CHANGESLICE is undoable, these events are recoverable. Consider the following example:

```
←PB LISPXHISTORY
@ TOP : ((& & --) 19 30 --)
←(CHANGESLICE 1 LISPXHISTORY)
1 is too small
←(CHANGESLICE 3 LISPXHISTORY)
3
←PB LISPXHISTORY
@ TOP : ((& & --) 23 3 --)
```

28.6.7 Searching the History List

HISTORYMATCH is used to match patterns against input portions of events on a history list in order to identify an event. It is primarily used by HISTORYFIND. It takes the form

```
Function:       HISTORYMATCH
# Arguments:    3
Arguments:      1) an input entry, INPUT
                2) a pattern, PATTERN
                3) an event specification, EVENT
Value:          As returned by EDITFINDP.
```

HISTORYMATCH is used by HISTORYFIND to match the pattern given by PATTERN against an entry from the specified history list. HISTORYFIND maps over the entries on the history list while calling HISTORYMATCH. It is initially defined as (EDITFINDP INPUT PATTERN T).

28.6.8 Printing the History List

PRINTHISTORY allows you to print a history list on a file. It takes the form

```
Function:       PRINTHISTORY
# Arguments:    5
Arguments:      1) a history list, HISTORY
                2) an event specification, EVENT
                3) a skip function, SKIPFN
                4) a values flag, NOVALUES
                5) a file name, FILE
Value:          A display of the events on the file.
```

PRINTHISTORY is used by the Programmer's Assistant command ?? to display the history list. You may display portions of the history list from within your program using it. Alternatively, it is useful if you keep your own history list for user interaction with a program since these events are not typed in at the top level and so are not recorded on LISPXHISTORY.

SKIPFNS is a functional argument (i.e., a LAMBDA expression) that is applied to each entry on the history list. If it returns a non-NIL value, that entry will not be printed.

If NOVALUES has the value T, the value will not be printed. This is usually the case when you want to display entries from EDITHISTORY.

Miscellaneous Functions and Features

INTERLISP provides a number of functions that seem to be orphans, i.e., they do not fit into any of the categories that we have already identified. This section describes these functions. In many cases, these functions are implementation dependent; they may not be supported in all INTERLISP systems.

In this text, I have omitted discussion of some of the functions that are implementation dependent. You should consult the INTERLISP Reference Manual for the proper calling sequences and parameters associated with these capabilities.

29.1 CHRONOMETRIC AND COUNTING FUNCTIONS

INTERLISP provides several functions for obtaining the current time and/or date. It also provides two functions for measuring the amount of time INTERLISP spends in performing certain types of computations.

29.1.1 Date and Time Functions

You may obtain the current date and time, formatted as a single string, by executing **DATE**, which takes the following form

```
Function:       DATE
# Arguments:    0
Arguments:      N/A
Value:          A string representing date and time.
```

DATE returns a string having the format

```
"dd-mm-yy hh:mm:ss"
```

```
where     dd   is the day
          mm   is the month
          yy   is the year
          hh   is the hour
          mm   is the minute
          ss   is the second.
```

The current date would appear as

```
←(DATE)
"28-OCT-83 07:17:34"
```

The date is obtained from the host operating system under which INTERLISP is implemented. The granularity of the time value is dependent on the accuracy of the clock maintained by the host operating system.

Obtaining the Date as a Large Integer

Another chronometric function is **IDATE**, which takes the form

```
Function:       IDATE
# Arguments:    1
Argument:       1) a date and time string, DT
Value:          An integer representation of DT.
```

IDATE converts DT to an integer representation. Given two data and time strings, DT1 and DT2, if DT1 is earlier than DT2, then IDATE[DT1] < IDATE[DT2]. If DT is NIL, IDATE returns the value associated with (DATE).

```
←(IDATE)
466167809
←(DATE)
"28-OCT-83 07:17:34"
```

A single integer value representing the date is useful for time-stamping entries in a list because it takes less space than a string.

Obtaining the Date and Time as a String

GDATE converts an internal date-and-time format into a date-time string. It takes the form

```
Function:       GDATE
# Arguments:    3
```

```
Arguments:      1) an internal date-and-time, DT
                2) a format, FORMATBITS
                3) a string pointer, STRPTR

Value:          A formatted date-time string.
```

The value of DT is equivalent to that produced by IDATE. If DT is NIL, then the value of (IDATE) is used.

```
←(GDATE)
"28-OCT-83 07:18:53"

←(GDATE (IDATE))
"27-Nov-84 13:02:28"

←(SETQ DT (IDATE))
12066947650

←(GDATE DT)
"27-Nov-84 13:03:11"
```

FORMATBITS is used by INTERLISP-10 to format the string using the TENEX/TOPS20 operating system routines.

Note that you may use IDATE to time-stamp a list element, but later recover the corresponding string using GDATE.

29.1.2 Clock Functions

Most INTERLISP systems can reference the time-of-day clock that is maintained by the system hardware. **CLOCK** returns various measures of time based on values extracted from the time-of-day clock. It takes the form

```
Function:       CLOCK

# Arguments:    1

Argument:       1) a format indicator, N

Value:          As described below.
```

The value of N indicates which of four possible measures will be returned by CLOCK:

0

The current value of the time-of-day clock in milliseconds. This value is equivalent to the number of milliseconds that have elapsed since the system was started up.

```
←(CLOCK)
-1983526334
```

1
The number of milliseconds since this session of INTERLISP was started.

```
←(CLOCK 1)
-1990867982
```

2
The number of milliseconds of compute time since this INTERLISP was started (garbage collect time is subtracted off).

```
←(CLOCK 2)
2311790
```

3
The number of milliseconds of compute time spent in garbage collection.

```
←(CLOCK 3)
268915
```

The values returned by CLOCK are not valid inputs to GDATE.

```
←(GDATE (CLOCK))
''9-May-1861 06:49:39''
```

29.2 SYSTEM FUNCTIONS

Because INTERLISP is usually implemented as an applications program running under control of an operating system, it provides a number of system functions that allow you to interact with the external environment. Functions described in this section are very dependent upon the nature of the underlying operating system. The values used in the examples are drawn from INTERLISP running on a Xerox 1100 Scientific Information Processor or INTERLISP-10 running on a DECSystem-20.

29.2.1 Exiting INTERLISP

To exit INTERLISP, you use the function **LOGOUT**, which takes the form

```
Function:       LOGOUT
# Arguments:    1
Argument:       1) a fast exit indicator, LOGOUTFLG
Value:          Meaningless.
```

LOGOUT exits INTERLISP and returns control to the underlying operating system. If LOGOUTFLG is T, INTERLISP exits without updating the virtual memory file that is maintained on the external disk. Thus, you probably will not be able to restart INTERLISP without first initializing it. (This fact is guaranteed under INTERLISP-D!). If LOGOUTFLG is NIL (as is usually the case), INTERLISP updates the virtual memory file by writing "dirty pages" from memory to the disk before returning control to the operating system.

LOGOUT has no value since it never returns to the user. However, if you restart INTERLISP using the virtual memory stored on disk, it will resume operation after the LOGOUT function invocation which will have the value NIL. Your program may continue from this point as if nothing had happened, save for the difference in time.

29.2.2 Obtaining the System Type

Because implementations of INTERLISP differ for various systems, it may be necessary for your program to know which system it is running on. **SYSTEMTYPE** returns an atom which indicates the current system on which you are running. It takes the form

Function:	SYSTEMTYPE
# Arguments:	0
Arguments:	N/A
Value:	A literal atom corresponding to the system type.

Consider the following examples:

```
←(SYSTEMTYPE)
D                "indicating a Xerox 1100 (a.k.a.
                 Dolphin)"

←(SYSTEMTYPE)
TOPS20           "indicating a DECSystem-20 computer"
```

You may use the value returned by SYSTEMTYPE to determine what code to execute or compile. This is most easily coded using the form

```
(SELECTQ (SYSTEMTYPE)
     (TOPS20 ...)
     (D      ...)
etc.
```

29.2.3 Obtaining the User Name

You may configure your program or your INTERLISP environment to operate differently depending on the user who is logged into it. **USERNAME** takes the form

```
Function:       USERNAME
# Arguments:    2
Argument:       1) a directory flag, DFLAG
                2) a type flag, TFLAG
Value:          As described below.
```

If DFLAG is NIL, USERNAME returns the name of the login directory owner. For Xerox 11xx systems, this name is written into the disk partition directory when the disk is initialized.

```
←(USERNAME)
"GOLDEN TIGER"
```

If DFLAG is T, USERNAME returns the name of the currently connected directory.

```
←(USERNAME 'T)
[100#]〈KAISLER〉LISP
```

TFLAG determines what type the value is returned as. If TFLAG is NIL, the value is returned as a string. If it is T, the value is returned as a literal atom.

```
←(USERNAME NIL T)
KAISLER
```

TFLAG may also be a string pointer whence the new string value is smashed into the location referenced by the pointer.

29.3 PERFORMANCE MEASURING FUNCTIONS

INTERLISP provides several functions for measuring the performance of your program in order to tune it. These functions are implementation-dependent. Their usage is described here for INTERLISP-10, but they will be discussed for INTERLISP-D in volume 2.

29.3.1 Counting CONS Operations

CONSCOUNT returns the number of CONSes since INTERLISP started up. Since all of the basic operations in INTERLISP ultimately become CONS operations, it may be used as a measure of system efficiency. It takes the form

```
Function:       CONSCOUNT
# Arguments:    1
Argument:       1) a new value, N
Value:          The current CONS count.
```

If N is non-NIL, CONSCOUNT sets the CONS counter to N after returning its current value. Typically, N is 0, so that you may measure the number of operations that a single expression or a function execution requires.

```
←(CONSCOUNT 0)
611962
←(CONSCOUNT)
15
```

29.3.2 Counting Page Faults

PAGEFAULTS determines the number of page faults that have occurred in your INTERLISP system. Originally, it was intended for use with earlier versions of INTERLISP where physical memories were a few hundred thousand bytes or so. However, even with the current size of memories, it is still a useful function for determining something about the efficiency of your implementation and program construction. It takes the form

```
Function:       PAGEFAULTS
# Arguments:    0
Arguments:      N/A
Value:          An integer representing the number of page
                faults.
```

PAGEFAULTS returns a number that is the number of page faults that have occurred since INTERLISP was started up.

```
←(PAGEFAULTS)
8717
```

Using PAGEFAULTS requires some knowledge about how your program is constructed. When you load a file, new atoms are allocated for the names of objects in the file, and memory is allocated for their values and function definitions. This allocation occurs in virtual memory. If you load a large number of files, virtual memory allocations will exceed physical memory. Excessive page faults require additional disk reads and writes, thus reducing the efficiency of your application.

29.3.3 Timing an Expression

You may compute the execution time of an expression using the function **TIME**, which takes the form

```
Function:       TIME
# Arguments:    3
Arguments:      1) an expression to be timed, TIMEX
                2) a repetition factor, TIMEN
                3) the type of timing, TIMETYPE
Value:          The value of the last execution of TIMEX.
```

TIME is an NLAMBDA function. It evaluates the expression TIMEX and prints out the number of CONSes and the computation time. Garbage collection time is subtracted out. Consider the following example (under INTERLISP-10):

```
←(SETQ PRESIDENTS '(FORD NIXON REAGAN HOOVER MCKINLEY))
(FORD NIXON REAGAN HOOVER MCKINLEY)

←(TIME (CAR PRESIDENTS))
0 conses
.001 seconds
FORD

←(TIME (LAST PRESIDENTS))
0 conses
.006 seconds
(MCKINLEY)

←(TIME (APPEND PRESIDENTS (LIST 'CARTER)))
6 conses
.005 seconds
(FORD NIXON REAGAN HOOVER MCKINLEY CARTER)
```

If TIMEN is greater than 1, TIMEX will be executed TIMEN times. TIMEN equal to NIL is assumed to be mean 1. The resulting number of conses and computation times are divied by TIMEN to produce an average over multiple cases. Consider the following example:

```
←(TIME (COPY PRESIDENTS) 20)
100/20 = 5 conses
.016/20 = .0008 seconds
(FORD NIXON REAGAN HOOVER MCKINLEY)
```

TIMETYPE controls the printing of additional information regarding the execution of TIMEX. If TIMETYPE has the value 0, TIME will measure and print the total real time as well as the computation time.

```
←(TIME (LOAD 'COMPLEX) NIL 0)
⟨KAISLER⟩COMPLEX..6
FILE CREATED 21-Nov-84 20:27:17
COMPLEXCOMS
1524 conses
2.041 seconds
6.845 seconds, real time
```

If TIMETYPE has the value 3, TIME will measure and print the garbage collection time as well as the computation time.

```
←(TIME (LOAD 'COMPLEX) NIL 3)
⟨KAISLER⟩COMPLEX..6
FILE CREATED 21-Nov-84 20:27:17
COMPLEXCOMS
1509 conses
1.922 seconds
.526 seconds, garbage collection time
```

If TIMETYPE has the value T, TIME will measure and print the number of page faults.

29.3.4 Breaking Down Performance by Function

You may break down the statistics collected by TIME on a function by function basis using **BREAKDOWN**, which takes the form

```
Function:       BREAKDOWN
# Arguments:    1-N
Arguments:      1-N) function names, FN[1] ... FN[n]
Value:          A list of functions that have been timed.
```

BREAKDOWN is an NLAMBDA, nospread fucntion. Its arguments are a list of function names which are modified to accumulate various statistics. When you have finished monitoring the performance of these functions, you may remove them from the monitoring process via UNBREAK. Each time you call BREAKDOWN, new functions may be added to the list of functions that are being monitored and the counters for all functions currently on the list are set to zero. BREAKDOWN takes care not to "doublecharge" time to two functions. This may occur when two functions are to be measured and one calls the other.

Displaying the Results of BREAKDOWN

The results accumulated by BREAKDOWN may be analyzed and displayed by **BRKDWNRESULTS**, which takes the form

```
Function:       BRKDWNRESULTS
# Arguments:    1
Arguments:      1) a return values flag, RETVALFLAG
Value:          Depends on the value of RETVALFLAG.
```

BRKDWNRESULTS will print an analysis of the statistics and number of calls to each function noted on the monitoring list. If RETVALFLAG is non-NIL, BRKWDNRESULTS does not print the analysis, but returns the values in a list whose entries are

```
(⟨function⟩ ⟨#calls⟩ ⟨value⟩)
```

Consider the following example:

```
←(BREAKDOWN CMULT TIMES REAL IMAG PLUS DIFFERENCE
COMPLEX)
(CMULT TIMES REAL IMAG PLUS DIFFERENCE COMPLEX)
←(SETQ CX1 (COMPLEX 1.0 3.0))
((1.0 . 3.0))
←(SETQ CX2 (COMPLEX 4.0 10.0))
((4.0 . 10.0))
←(SETQ CX3 (CMULT CX1 CX2))
((-26.0 . 22.0))
←(BRKDWNRESULTS)
FUNCTION        TIME      # CALLS   PER CALL  %

CMULT           .011      1         .011      2
TIMES           .022      4         .006      3
REAL            .097      7         .014      14
IMAG            .038      7         .005      5
PLUS            .006      2         .003      1
DIFFERENCE      .004      1         .004      1
COMPLEX         .085      3         .028      12
SETQ            .430      4         .107      62
TOTAL           .693      29        .024
NIL
←(BRKDWNRESULTS T)
((CMULT 1 11) (TIMES 4 22) (REAL 7 97) (IMAG 7 38) (PLUS
2 6) (DIFFERENCE 1 4) (COMPLEX 3 85) (SETQ 4 430))
```

BREAKDOWN may also be used to gather other statistics by setting the value of the variable BRKDWNTYPE prior to calling BREAKDOWN. The

types of statistics that may be gathered are given by the value of BRKDWNTYPES:

```
←(PRINTDEF BRKDWNTYPES)
((TIME (CPUTIME)
       (LAMBDA (X)
               (FQUOTIENT X 1000)))
 (CONSES (CONSCOUNT))
 (PAGEFAULTS (PAGEFAULTS))
 (BOXES (IBOXCOUNT))
 (FBOXES (FBOXCOUNT))
```

Each entry should have the form

(⟨type⟩ ⟨expression⟩ ⟨function⟩)

where ⟨type⟩ is a statistic name
⟨expression⟩ is an S-expression that computes the statistic
⟨function⟩ may convert the statistic value to some more interesting quantity or format

By setting BRKDWNTYPE to one or more of the quantities listed above, BREAKDOWN makes the necessary changes to gather the statistics. The functions to be monitored will be redefined to gather the appropriate statistic.

Some of the functions that you are analyzing may be fast enough that the overhead involved in measuring the functions obscures the actual time spent in the function. In a manner similar to TIME, you may specify a value in BREAKDOWN that specifies that a function will be executed more than once.

```
←(BREAKDOWN 10 CMULT TIMES REAL IMAG DIFFERENCE PLUS
COMPLEX)
(CMULT TIMES REAL IMAG DIFFERENCE PLUS COMPLEX)

←(SETQ CX3 (CMULT CX1 CX2))
((-26.0 . 22.0))

←(BRKDWNRESULTS)
FUNCTIONS      TIME      # CALLS   PER CALL  %
CMULT          .189      1         .189      24
TIMES          .796      40        .020      103
REAL           1.718     41        .042      223
IMAG           1.749     41        .043      227
DIFFERENCE     .150      10        .015      19
PLUS           2.103     110       .019      272
COMPLEX        1.013     10        .101      131
TOTAL          .772      253       .003
NIL
```

29.4 SESSION TRANSCRIPTS

Your interaction with INTERLISP, from the time you sign on until you logout, is known as a session. During the session, you will execute many functions and programs. Many times you will want to retain a transcript (e.g., a copy) of your session. The transcript records all you type in to INTERLISP and all that it types back. (Note: In INTERLISP-D this does not include output from certain user packages such as GRAPHER.)

To initiate transcription, you execute the function **DRIBBLE**, which takes the form

```
Function:       DRIBBLE
# Arguments:    2
Arguments:      1) a file name, FILE
                2) a flag, APPENDFLG
Value:          Previous transcript file name.
```

DRIBBLE opens FILE and records what you type in and what INTERLISP types out in the file. If FILE is non-NIL, the current transcript, if any, is closed and a new one opened. DRIBBLE returns the full name of the previous transcript file.

To terminate transcription, you can set FILE to NIL. DRIBBLE closes the transcript file and returns its full name. No further information is recorded.

You may make a continuous transcript over several sessions by setting APPENDFLG to T. Thus, when you resume a session at a later time when FILE already exists, new information will be appended to FILE.

There are two caveats concerning transcript files:

1. A transcript file does not appear in the list of files that are returned by OPENP because it "shadows" your interaction with INTERLISP.
2. A transcript file will not be closed by either CLOSEALL or CLOSEF for the same reason. Only DRIBBLE can close a transcript file.

Obtaining the Current Dribble File Name

In some programs, you may want to switch among transcript files depending on the current task you're working on. To do so, you need to know the current transcript file name. **DRIBBLEFILE** returns the name of the current transcript file. It takes no arguments.

The following function suggests a way to do this switching:

```
(DEFINEQ
   (select-transcript-file (task)
 (COND
    ((EQUAL (DRIBBLEFILE)
```

```
            (task-transcript-file task)))
      (T
          (SELECTQ task
             ((task-id1
                (DRIBBLE (task-transcript-file task) T))
                          .
                          .
                          .
             (PROGN
                (DRIBBLE NIL)))))
))
```

where TASK-TRANSCRIPT-FILE is your function to determine the name of the transcript file associated with the particular task. Note that each time you open the transcript file associated with the task you append new information to it.

Using this approach, you must define the names of the transcript files associated with each task at the beginning of your session. At the end of the session, you must be sure to close the current transcript file.

29.5 GREETINGS

You may personalize your INTERLISP environment in several different ways. One way is to set up a greeting for yourself or other users. Greetings are stored in a file INIT.LISP that resides in your directory.

When INTERLISP is initiated, it looks for two INIT files:

1. The first INIT file is looked for in the LISP directory. It is usually a site-specific profile that supports mutiple users at a site.
2. The second INIT file is a user-specific file. It contains expressions that tailor your environment. It should be located in your directory.

INTERLISP loads the respective INIT files, if found, and executes the expressions contained therein (see Section 29.6).

INTERLISP prints a greeting using the value of FIRSTNAME which should be set in your INIT file via

```
(INITVARS (FIRSTNAME "Steve"))
```

or

```
(E (SETQ FIRSTNAME "Steve"))
```

Changing the Greeting

You can change the form of the greeting in two ways. First, you may define your own function, **GREET**, which is invoked by INTERLISP when you logon. Your

definition, which should be loaded via LOAD or defined by DEFINEQ in your INIT file, will replace the standard greeting displayed by INTERLISP.

Second, you can specify greetings for specific dates by modifying the variable GREETDATES. GREETDATES is a list of elements of the form (⟨datestring⟩ . ⟨string⟩). New entries may be added to GREETDATES by including expressions of the following form in your INIT file:

```
(ADDVARS (GREETDATES (CONS ⟨string1⟩ ⟨string2⟩)))

(ADDVARS (GREETDATES (CONS "25-DEC" "Merry Christmas")))
(ADDVARS (GREETDATES (CONS "16-OCT" "Happy Birthday")))
```

29.6 DIRECTORY ACCESS FUNCTIONS

Most of your files will reside in a directory which is a collection of files belonging to a single user or project. At any time, you are connected to one directory although you may access files in many other directories by prepending the directory name to the file name. INTERLISP provides a number of functions for accessing directories from within INTERLISP.

29.6.1 Reading the File Directory

You may read the file directory of the external disk using **FILDIR**, which takes the form

```
Function:       FILDIR
# Arguments:    1
Argument:       1) a file group specification, FILGROUP
Value:          A list of files matching the
                specification.
```

FILDIR reads the current directory on the external disk and returns a list of files that match the file group specification. The value of FILGROUP depends on the file name representation used by the operating system.

```
←(FILDIR '*.DCOM)
([DSK]BROWSER.DCOM;1 [DSK]HISTMENU.DCOM;1
[DSK]MAKEGRAPH.DCOM;1   [DSK]UTILPROC.DCOM;1)
```

which locates all files having the extension ".DCOM".

29.6.2 Manipulating File Directories

You may manipulate a file directory using **DIRECTORY**, which takes the form

Function:	DIRECTORY
# Arguments:	4
Arguments:	1) a files specification, FILES 2) a set of commands, COMMANDS 3) a default text, DEFAULTEXT 4) a default version, DEFAULTVER
Value:	A list of all files that satisfy the commands given to DIRECTORY.

FILES is a files specification that identifies the files to be operated upon. It may take one of three forms:

1. NIL, whence the form *.*;* is assumed meaning all files.
2. An atom containing ⟨ESC⟩s or *s which match any number of characters or ?s which match a single character.
3. A list of one the forms

```
(⟨files⟩ + ⟨files⟩) "and"
(⟨files⟩ - ⟨files⟩) "and not"
(⟨files⟩ * ⟨files⟩) "or"
```

For example, (T$ + $L) will match any file name in the directory that begins with T or ends in L, while (T$ — *.DCOM) will match all files that begin with T and are not .DCOM files.

For each file that matches, the commands in COMMANDS are executed. The available commands are

P	Print the file name
PP	Print the file name (except the version number)
⟨string⟩	Print the string
READDATE WRITEDATE CREATIONDATE SIZE LENGTH BYTESIZE PROTECTION AUTHOR TYPE	Print the appropriate information returned by GETFILEINFO (see Section 16.4)

COLLECT	The value returned by DIRECTORY will be a list of file names; add the complete file name to this list.
COUNTSIZE	The value of DIRECTORY will be a sum; add the size of this file to that sum.
PAUSE	Wait until the user types a character before proceeding with the rest of the commands; this allows you to display some information and think about it.
PROMPT ⟨message⟩	Prompt the user with ⟨message⟩. If he responds NO, abort command processing for this file.
OLDERTHAN ⟨n⟩	Continue command processing if the file hasn't been referenced in N days.
OLDVERSIONS ⟨n⟩	Continue command processing if there are at least N more recent versions of the same file. Usually, this ensures that you have an adequate audit trail for the file within the directory. Used primarily with DELETE.
BY ⟨user⟩	Continue command processing if the file was written by a specified user.
DELETED	Examine files that are marked deleted within the directory as well.
OUT ⟨file⟩	Print the results to the specified output file.
COLUMNS ⟨n⟩	Attempt to format the output in N columns, rather than just 1.
TRIMTO ⟨n⟩	Delete all but N versions of the file; N must be greater than or equal to 0.
DELETE	Delete the file. If this command is specified without COLLECT, the value of DIRECTORY will be NIL; otherwise, a list of files deleted.
UNDELETE	Undeletes the indicated files that have just been deleted.

Consider the following examples:

```
←(DIRECTORY '⟨LISPUSERS⟩*.COM 'PP)
⟨LISPUSERS⟩
ALL.COM
ARITHDECLS.COM
ARITHMAC.COM
BQUOTE.COM
CHARCODE.COM
CHAT.COM
...
←(DIRECTORY '⟨LISPUSERS⟩S$.* '(PP SIZE))
    ⟨LISPUSERS⟩
SAMEDIR.            2
SAMEDIR.COM         0
SAMEDIR.TTY         2
SCRATCHLIST.        0
SCRATYCHLIST.COM    0
SCREENOP.           28
SCREENOP.COM        29
SCREENOP.TC         14
SCREENOP.TCD        12
SCREENOP.TTY        2
...
```

DIRECTORY uses **DIRCOMMANDS** to correct spelling and to define synonyms and abbreviations for commands. The structure of **DIRCOMMANDS** is

```
←(PRINTDEF DIRCOMMANDS)
((- . PAUSE)
 (AU . AUTHOR)
 BY COLLECT (COLLECT? PROMPT " ? " COLLECT)
 COUNTSIZE
 (DA . WRITEDATE)
 (DEL . DELETE)
 (DEL? . DELETE?)
 DELETE
 (DELETE? PROMPT "delete? " DELETE)
 DELETED
 (LE LENGTH "(" BYTESIZE ")")
 (OBS . OLDVERSIONS)
 OLDVERSIONS
 (OLD OLDERTHAN 90)
 OLDERTHAN
 (OU . OUT)
```

```
OUT P PAUSE (PR . PROTECTION)
PROMPT
(SI . SIZE)
(TI . WRITEDATE)
UNDELETE
(VERBOSE AUTHOR CREATIONDATE SIZE READDATE WRITEDATE))
```

29.6.3 Connecting to Another Directory

You will note in the examples above that I had to specifically provide an alternative directory name that I wanted to inspect. You may connect to a directory (i.e., make it your current directory) using **CNDIR**, which takes the form

```
Function:       CNDIR
# Arguments:    2
Arguments:      1) the directory name, DIRNAME
                2) the directory password, PASSWORD
Value:          The name of the old directory.
```

Under INTERLISP-10, CNDIR resides in the LISPUSERS package EXEC-.COM. You must load that package first in order to execute CNDIR. Consider the following example:

```
←(LOAD '⟨LISPUSERS⟩EXEC.COM)
⟨LISPUSERS⟩EXEC.COM.54
FILE CREATED 13-JUN-82 15:27:46
collecting arrays
1071, 10287 free cells
EXECCOMS
⟨LISPUSERS⟩PASSWORDS.COM.15
compiled on 6-FEB-82 20:19:30
FILE CREATED 6-FEB-82 20:19:17
PASSWORDSCOMS
⟨LISPUSERS⟩EXEC.COM.54
←(CNDIR 'LISPUSERS)
PS:⟨LISPUSERS⟩ (password) "I don't know the password so I
                          just type a carriage return"
Can't connect to directory
```

If a password is associated with a directory and is not provided in the call to CNDIR, you will prompted for the password.

```
←(CNDIR)
PS:<KAISLER>
```

If DIRNAME is NIL, CNDIR just prints the name of your current directory.

29.7 STORAGE MANAGEMENT

INTERLISP eases the burden of storage management by doing dynamic allocation for various datatypes on demand as you create new objects. In INTERLISP-D, you usually do not have to worry about how much storage you have, because garbage collection is performed on an incremental basis. However, when programs get very large and complex, you may find that you have to sacrifice one type of storage in order to accommodate more objects of another type. This situation occurs because INTERLISP divides available memory into a number of pools associated with different object types. When a pool is exhausted, it may be increased from an available space pool.

29.7.1 Displaying Storage Usage

STORAGE provides you with a description of storage usage within the INTERLISP environment. It takes the form

```
Function:       STORAGE
# Arguments:    0
Arguments:      N/A
Value:          NIL
```

The "actual" value of STORAGE is a formatted listing that is written to the primary output file.

The following example is taken from the Fugue release running on a Xerox 1100 Scientific Information Processor.

```
←(STORAGE)
```

Type	Assigned	Free Items	In Use	Total Allocations	Pages (items)
FIXP	20	2560	1752	808	26271
FLOATP	14	1792	537	1255	1577
LISTP	1512	175392	1419	137973	578679
ARRAYP	8	512	326	186	10527
STRINGP	68	4352	119	4233	7080
STACKP	4	512	504	8	1894

```
VMEMPAGEP       106       106         3        103         104
STREAM            6        33        14         19          19
FDEV              6        24         5         19          19
IMAGEOPS
BitBltTable
TERMTABLEP
READTABLEP                      <more numbers>
CHARTABLE
IOFILEINFOBLK
BUFFER
PROCESS
PROCESSQUEUE
EVENT
MONITORLOCK
SYSQUEUE
...
TOTAL           1910

Data Spaces     Allocated    Remaining
                  Pages        Pages

MDS               1910         1674
Atoms               82           46
Print Names        507         1541
Arrays            2817         4095
NIL
```

29.7.2 Gaining Space

When your program gets too large, INTERLISP will start to complain. The nature of its complaint depends on the implementation. INTERLISP-D stores its virtual memory on the integral disk of the Xerox 1100 Scientific Information Processor. When the initial allocation (which Xerox recommends to be contiguous space) is exhausted, INTERLISP grabs the largest remaining chunk of free contiguous space that it can find. This is known as a *segment* of the virtual memory file. Since segments are disconnected, performance tends to degrade rapidly. At some point, an internal decision is made that there are too many segments. INTERLISP warns you that you must reconfigure your system (particularly your virtual memory file) before it can continue. You are given a certain grace period in which some curative steps may be taken.

GAINSPACE is a function that interacts with you to remedy the problem of exhausted space. You probably should treat it as the method of last result because the actions it takes are generally disruptive of the environemt—by that, I mean that it disables considerable subsystems of INTERLISP in order to give you the space that you require.

GAINSPACE takes the following form

```
Function:       GAINSPACE
# Arguments:    0
Arguments:      N/A
Value:          NIL
```

When it is invoked, it conducts a dialogue with you to determine what features you are willing to give up. The side effect of GAINSPACE is to reclaim storage space through several rather drastic measures. It is usually wise to execute STORAGE after running GAINSPACE to determine what has happened.

```
←(GAINSPACE)
erase current Masterscope database? Yes
discard HPRINT initialization? Yes
purge History lists? Yes
discard definitions on property lists? Yes
discard old values of variables? Yes
erase properties? No
erase CLISP translations? Yes
erase system hash array? Yes
discard context of last edit? Yes
discard information saved for undoing your greeting? Yes
erase filepkg information? No
mapatoms called to erase the indicated properties...done
```

GAINSPACE uses the list GAINSPACEFORMS to drive its dialogue. Each entry on GAINSPACEFORMS has the format

```
(⟨precheck⟩ ⟨message⟩ ⟨form⟩ ⟨keylst⟩)
```

PRECHECK is evaluated. If it returns NIL, GAINSPACE skips to the next entry. Otherwise, ASKUSER (see Section 14.7) is called with MESSAGE and the optional KEYLST. If the user responds **N** (the "o" is filled out by INTERLISP), i.e., ASKUSER returns N, GAINSPACE moves to the next entry. Otherwise, it evaluates FORM with RESPONSE bound to the value returned by ASKUSER.

GAINSPACEFORMS has the following value:

```
((MSDATABASELST
     "erase current Masterscope database"
     (% . ERASE))
     ((OR HPRINTHASHARRAY HPRINTRDTBL)
```

```
        "discard HRPINT initialization"
        (PROGN
             (CLRHASH HPRINTHASHARRAY)
             (SETQ HPRINTHASHARRAY (SETQ
             HPRINTRDTBL))))
   ((CAR LISPXHISTORY)
        "purge history lists"
        (PURGEHISTORY RESPONSE)
        ((Y "es")
         (N "o")
         (E "verything")))
   (T
        "discard definitions on property lists"
        (SETQ SMASHPROPSLST1
              (CONS 'EXPR
                    (CONS 'CODE
                          (CONS 'SUBR
                          SMASHPROPSLST1)))))
   (T
        "discard old values of variables"
        (SETQ SMASHPROPSLST1
              (CONS 'VALUE SMASHPROPSLST1)))
   (T
        "erase properties"
        (ERASEPROPS RESPONSE)
        ((Y "es"
            EXPLAINSTRING
            "Yes-you will be asked which properties are
            to be erased"
         (N "o")
         (A "ll"
            CONFIRMFLG T EXPLAINSTRING
            "All-all properties mentioned on
            SMASHPROPSMENU")
         (E "dit"
            EXPLAINSTRING
            "Edit-you will be allowed to edit a list of
            property names")))
   (CLISPARRAY
        "erase CLISP translations"
        (CLRHASH CLISPARRAY))
   (CHANGESARRAY
        "erase changes array"
        (CLRHASH CHANGESARRAY))
   (SYSHASHARRAY
```

```
     "erase system hash array"
     (CLRHASH))
((GETPROP 'EDIT 'LASTVALUE)
     "discard context of last edit"
     (REMPROP "EDIT "LASTVALUE))
(GREETHIST
     "discard information saved for undoing your
     greeting"
     (SETQ GREETHIST))
(FILELST
     "erase filepkg information"
     (CLEARFILEPKG RESPONSE)
     ((Y "es")
      (N "o')
      (E "verything')
      (F "ilemaps only"))))
```

SMASHPROPSMENU has the following form:

```
(("old values of variables"
     VALUE)
 ("function definitions on property lists"
     EXPR CODE)
 ("advice information"
     ADVISED ADVICE READVICE
     (SETQ ADVISEDFNS NIL))
 ("filemaps"
     FILEMAP)
 ("clisp information (warning: this will disable
 clisp!)"
     ACCESSFN BROADSCOPE CLISPCLASS CLISPCLASSDEF
     CLISPFORM
     CLISPIFYISPR CLISPINFIX CLISPISFORM CLISPISPROP
     CLISPNEG CLISPTYPE CLISPWORD CLMAPS I.S.OPR
     I.S.TYPE LISPFN SETFN UNARYOP)
 ("compiler information (warning: this will disable
 the compiler!)"
     AMAC BLKLIBRARYDEF CROPS CTYP GLOBALVAR MACRO
     MAKE OPD UBOX)
 ("definitions of named history commands"
     *HISTORY*)
 ("context of edits exited via save command"
     EDIT-SAVE))
```

The INTERLISP Execution Environment

INTERLISP is based on the principle of the *stack* or pushdown list. Each function is allocated a *frame* on the stack when the function is invoked. The frame establishes the *environment* of the function, namely, its local variable bindings and argument values, if any. INTERLISP provides many low-level functions for inspecting and manipulating the stack.

Caution:
Only sophisticated users should attempt to manipulate the stack. Erroneous changes to the stack can destroy your computation's environment and erase all the work you have done.

30.1 BINDING OF VARIABLES

Many schemes have been used to bind values to variables. INTERLISP uses one of two different schemes depending on the implementation you are using:

Shallow Binding
Values for variables are stored in value cells associated with the variable name. When a function is entered, the variables it uses are rebound. Values stored in the value cells are stored in a stack frame associated with the function call. When the function is exited, each variable must be individually unbound, i.e., its values in the stack frame are restored to the value cells. INTERLISP-10, INTERLISP/370, and INTERLISP-VAX use shallow binding.

Deep Binding
When a variable is bound, space is allocated on the stack for the variable name and its value cell. The variable's value is placed in the value cell. When a variable is accessed, its value is found by searching the stack from the top for the most recent occurrence of the variable name. The value is retrieved from the

associated value cell. If the variable is not found on the stack, the variable's top-level value cell is accessed for the value. INTERLISP-D uses deep binding.

30.1.1 Variable Types

INTERLISP supports three types of variables:

LOCALVARS
A *local variable* is one that is used within a function, LAMBDA/NLAMBDA expression, or a PROG. The variable's binding exists only for the duration of the function's or PROG's execution.

GLOBALVARS
A *global variable* is a "free" variable, e.g., one that has not been bound by any function. Its binding is always stored in the top-level value cell of the atom. The variable's binding exists for the duration of your sysout.

SPECVARS
A *special variable* is a variable referenced in a function that was bound in another function or bound globally. Basically, it means the variable is bound outside the function but is visible within the function. The difference between SPECVARS and LOCALVARS mainly has to do with how the variables are accessed.

30.1.2 Global Variables

Global variables may be specified in INTERLISP in two ways:

1. Putting the variable name on the GLOBALVARS list
2. Putting the property GLOBALVAR with value T on the variable's property list.

Global variables are always accessed via their top-level value when they are used freely in a function (using GETTOPVAL—see Section 3.9.2). Values are assigned to global variables using SETTOPVAL even when SETQ is invoked by your program.

INTERLISP-D uses a deep-binding scheme while INTERLISP-10/VAX/370 all use a shallow-binding scheme. In deep-binding, references to variables use the nearest binding for the variable in the stack to the function where the variable is referenced. If the variable is passed as an argument to the function, then it is bound in the function's stack frame. Otherwise, it is used freely and INTERLISP must search the stack for the binding. Searching may be expensive if the nesting of function calls is extensive. To circumvent the searching you may use global variables to provide more efficient access to the variable's value.

In a shallow-binding scheme, no distinction is made between global variables and other variable types. All variable references access the variable's value cell. Access to a variable's value is always independent of the depth of the stack.

INTERLISP-10/VAX/370 run on general-purpose computers which usually have instructions conducive to executing conventional languages such as FORTRAN. In fact, the DECSystem-10 and IBM 370/308x/43xx series machines do not have hardware stack support, so shallow binding provides a greater measure of efficiency. On the other hand, INTERLISP-D runs on the custom-built Xerox 11xx Scientific Information Processors which provide enhanced hardware stack support mechanisms.

Global variables are treated like SPECVARS when function are compiled. That is, their names are always placed on the stack when they are rebound.

INTERLISP treats all system parameters, unless otherwise specified, as global variables. Rebinding these variables in a deep-binding scheme does not affect the behavior of the system because that binding will be placed in a function's stack frame. To affect the system's behavior, you must use RESETVAR (see Section 25.7.3) to establish a new value.

30.2 STACK STRUCTURE

A stack is a pushdown list. A discussion of stack algorithms may be found in [knut68]. INTERLISP uses a stack structure known as a *spaghetti stack* which is described in [bobr73a]. A detailed discussion of the INTERLISP-10 stack mechanisms at the virtual machine level may be found in [moor79].

30.2.1 A Basic Frame Example

Using BTV! in the Break Package, we can display the basic frame of the function MAKESLOT as it appears under INTERLISP-D (Fugue release) when the function is newly broken. The data are taken from a Xerox 1100 Scientific Information Processor.

```
(MAKESLOT broken)
:BTV!
Basic frame at 40174
40160:          0    51655     NODE TEXAS
40162:          0    52021     SLOT CAPITOL
40164:          0    15071     INHERITANCE.TYPE IS
40166:          0      113     METHOD S
40170:          0    51513     *local* MAKESLOT
40172:          0        0     [padding]
40174:     100400    40160
```

30.2.2 A Frame Extension Example

A frame extension is a variable-length block of storage containing

a frame name
a pointer to variable bindings, BLINK
pointers to other frame extensions, ALINK/CLINK
temporary variables
reference counts

BTV! also prints the contents of the frame extension for MAKESLOT as shown below:

```
Frame xtn at 40176 frame name = MAKESLOT
40176:    141002    40130     [V, USE=2, alink]
40200:    103524    20040     [fn header]
40202:     40332      412     [next, pc]
40204:     40300    13427     [nametable]
40206:    177777        0     [blink, clink]
40210:        16        5     *local* 5
40212:         0    51513     *local* MAKESLOT
40214:        16        4     *local* 4
40216:        16        4     *local* 4
40220:         4   135406     *local* (LAMBDA ...)
                              [definition of MAKESLOT]
40222:        10   135272     *local*    (NODE    SLOT
                              INHERITANCE.TYPE METHOD)
40224:        16        0     *local* 0
40226:        16       10     *local* 8
40230:        16       15     *local* 13
40232:         0    15714     *local* METHOD
40234:         0        0     *local* NIL
40236:         0        0     *local* NIL
40240:         0        0     *local* NIL
40242:    177777        0     *local* [unbound]
          through
40264:    177777        0     *local* [unbound]
40266:    177777        0     [padding]
40270:         0    51513     [padding]
40272:         0        0     [padding]
40274:    177776        0
40276:    177763       30
40300:         0        0     NIL
40302:         0        0     NIL
40304:         0    51513     MAKESLOT
```

```
40306:        10     13400     ([]#0,77400, []#60,174060)
40310:     13621     51604
40312:     51547     15714
40314:         0         0     NIL
40316:         0         0     NIL
40320:         0         1     NOBIND
40322:         2         3     [VMEMPAGE]#2,3
40324:         0         0     NIL
40326:         0         0     NIL
40330:         0         0     NIL
MAKESLOT
```

Two frame extensions may point to the same basic frame which allows two processes to communicate via shared variable bindings.

30.2.3 Stack Frames and Pointers

A stack is composed of *frames* which are allocations of storage to hold the values of local variables during computation. Coupled with frames are a data structure known as the *frame extension*. It is a variable-sized block of storage that contains

a frame name
a pointer to some variable bindings, BLINK
an access chain pointer, ALINK
a control chain pointer, CLINK
other implementation dependent information

Both ALINK and CLINK point to other frame extensions.

A frame extension completely specifies the variable bindings and control information necessary to evaluate a function.

At any instant, only one function is being executed. Thus, one frame will be distinguished as having control of the central processor. This frame is called the *active frame*. Initially, the top-level frame, which is hardwired into the interpreter, is the active frame. When a computation in an active frame invokes another function, a new basic frame and frame extension are built on the stack. The frame of the new basic frame will be the name of the function that is being invoked. The ALINK, BLINK, and CLINK fields of the new frame will be assigned values depending on the calling sequence to the function. The function is "run" by passing control to the new frame.

The chain of frame extensions that may be reached via the ALINK 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.

When an active computation has been completed in a frame, control usually returns to the frame pointed to by the CLINK. The frame in the CLINK becomes

the active frame. Thus, control of the computation is passed through the control chain. It can be said to reflect the progress of the computation. Tracing functions is effectively a process of backtracking through the control chain.

When a frame is exited, the storage associated with the basic frame and the frame extension may be reclaimed. You may hold on to a frame by creating a *stack pointer* to it. A stack pointer contains the address of the frame. This allows you to run other computations using that frame as an environment.

Stack pointers print in different formats, depending on the version of INTERLISP that you are running. They are returned by many of the stack functions described in the following sections. As long as a stack pointer has been allocated to a frame, that frame will not be garbage collected. Two stack pointers referencing the same frame are not necessarily EQ, but are EQP.

Allocating a stack pointer may consume a large amount of stack space. The space is not freed, even if it is no longer being used, until the next garbage collection. Thus, if you are using stack pointers rather vigorously, you may want to invoke garbage collection at opportune times during your computation.

If there is insufficient stack space to allocate a new frame, a STACK OVERFLOW condition will occur. Usually, there is not much you can do but reset the environment. Note that stack overflows may occur when you have runaway functions, particularly ones that are recursive.

30.3 STACK ACCESS FUNCTIONS

You may access the stack using the functions described in this section. Each stack function takes an argument which describes the stack frame that you want to access. The frame may be specified by

a frame name (e.g., MAKESLOT)

NIL, indicating the active frame (which is the stack frame of the stack function itself)

T, the top-level frame

a literal atom, which is equivalent to (STKPOS ⟨atom⟩ -1)

a number, which is equivalent to (STKNTH number)

Two additional errors may occur:

ILLEGAL STACK ARG
When a stack frame descriptor is expected and the argument supplied is not one of the above. Alternatively, if there is no frame corresponding to the argument.

STACK POINTER HAS BEEN RELEASED
When a released stack pointer is supplied as a stack frame descriptor for any purpose other than reuse.

The following functions convert names and numbers to pointers and names. They might be summarized as follows:

	TO	
	Function Name	*Stack Pointer*
FROM		
Function Name	N/A	STKPOS
Numbers	STKNTHNAME	STKNTH
Stack Pointer	STKNAME	N/A

30.3.1 Locating a Stack Frame

STKPOS allows you to locate a stack frame by its name. It takes the form

```
Function:       STKPOS
# Arguments:    4
Arguments:      1) a frame name, FRAMENAME
                2) an integer, N
                3) an initial position. IPOSITION
                4) an old stack pointer, OPOSITION
Value:          A stack pointer to the frame;
                otherwise, NIL.
```

STKPOS searches the stack for the Nth frame having the same name as FRAMENAME. If N is NIL, a −1 is assumed. The search begins at IPOSITION, which is a stack frame, and proceeds as follows:

along the control chain if N is negative
along the access chain if N is positive

If OPOSITION is non-NIL and is a stack pointer, it is re-used.

STKPOS is a generalized positioning algorithm. Argument N allows you to select the Nth of many similar frames. This feature is useful when you are searching the stack environment of a recursive function (a new frame is created for each recursive call), e.g.,

```
←(STKPOS 'MAKESLOT)
#1,13446/MAKESLOT
```

You are not allowed to create a stack pointer to the active stack frame. Thus,

```
←(STKPOS 'STKPOS)
ILLEGAL STACK ARG
```

Locating a Frame by Position

You may also locate a frame by providing its index from the top of the stack (which is the most recent function invocation). **STKNTH** takes the following form

```
Function:       STKNTH
# Arguments:    3
Arguments:      1) an integer, N
                2) an initial position, IPOSITION
                3) an old position, OPOSITION
Value:          A stack pointer to a frame or NIL.
```

STKNTH returns a stack pointer to the Nth frame from the top of the stack or, if IPOSITION is non-NIL, from IPOSITION. N is used as follows:

if N is negative, STKNTH follows the control chain
if N is positive, STKNTH follows the access chain
if N is 0, STKNTH returns a stack pointer to IPOSITION

Consider the following example:

```
←(STKNTH)
ILLEGAL STACK ARG
```

because you are trying to create a stack pointer to the active frame which is not allowed.

```
←(STKNTH 1)
#1,13452/APPLY
←(STKNTH 2)
#1, 13476/LISPX
```

because if we look at the stack, which we can inspect with the BT command from the Break Package, we see

```
(MAKESLOT broken)
:BT
```

```
MAKESLOT
\EVALFORM
\SAFEEVAL
BREAK1
\EVALFORM
EVAL
LISPX
ERRORSET
EVALQT
...
```

If OPOSITION is supplied and it is a stack pointer, it is reused; otherwise, it is ignored.

30.3.2 Obtaining and Changing the Frame Name

You may obtain the frame name for a stack position by executing **STKNAME**, which takes the form

```
Function:       STKNAME
# Arguments:    1
Arguments:      1) a stack descriptor, POSITION
Value:          The stack frame name.
```

STKNAME returns the name of the stack frame (i.e., the name of the function for which the frame was built) for the frame described by POSITION, e.g.,

```
←(STKNAME 2)
LISPX
←(STKNAME 1)
APPLY
←(STKNAME (STKPOS 'MAKESLOT))
MAKESLOT
```

because EVALQT is, effectively, an APPLY of LISPX to its inputs that are read by low-level LISP functions (See Section 25.2).

Obtaining the Nth Stack Frame Name

From an initial position, you may obtain the name of the NTH frame using **STKNTHNAME**. It takes the form

```
Function:       STKNTHNAME
# Arguments:    2
```

```
Arguments:      1) an integer, N
                2) an initial position, IPOSITION

Value:          The name of the stack frame or NIL.
```

STKNTHFRAME is equivalent to executing

```
(STKNAME (STKNTH N IPOSITION))
```

but avoids the creation of a stack pointer. Like STKNTH, it returns NIL if there are a lesser number of frames in the stack than N. Like STKNTH, it follows the control or access chains depending on the sign of N.

Changing a Stack Frame Name

You may change the name of a stack frame by executing **SETSTKNAME**, which takes the form

```
Function:       SETSTKNAME

# Arguments:    2

Arguments:      1) a stack position, POSITION
                2) a name, NAME

Value:          The new frame name.
```

Unless you are an experienced INTERLISP programmer, you should attempt to change the name of the stack frame. This function may be used when an error occurs.

30.4 VARIABLE BINDING FUNCTIONS

Variables appear as entries in a stack frame. You may change the binding of a variable or determine the binding using the following functions.

30.4.1 Obtaining Variables at a Stack Frame

VARIABLES returns a list of variables that are bound in the stack frame POSITION. It takes the form

```
Function:       VARIABLES

# Arguments:    1

Argument:       1) a stack frame, POSITION

Value:          The list of variables bound at the stack
                frame described by POSITION.
```

Consider the following example (from INTERLISP-10):

```
←(STKNTH 4)
[STACKP]#152002/ERRORSET
←(VARIABLES 4)
(HELPCLOCK LISPXHIST HELPFLG NIL LISPXLISTFLG LISPXLINE
NIL
NIL LISPXVALUE NIL NIL)
```

30.4.2 Obtaining Variable Values at a Stack Frame

STKARGS returns the values of the variables bound at POSITION. It takes the form

```
Function:       STKARGS
# Arguments:    1
Argument:       1) a stack frame, POSITION
Value:          A list of argument values.
```

Consider the following example (from INTERLISP-10):

```
←(STKARGS 'ERRORSET)
((DUMMY) T NIL)
```

30.4.3 Scanning the Stack for Bindings

When you are executing a complex program, your function nesting may become very deep. Within this nesting, some variables will be used freely while others will be bound from one function to another. If a function breaks, you may not know where a free variable is bound. **STKSCAN** searches the stack looking for a binding of a specified variable. It takes the form

```
Function:       STKSCAN
# Arguments:    3
Arguments:      1) a variable name, VAR
                2) an initial stack frame, IPOSITION
                3) an optional stack pointer, OPOSITION
Value:          A stack pointer to the frame in which VAR
                is bound; otherwise, NIL.
```

STKSCAN searches the stack from the stack descriptor specified by IPOSITION looking for a frame in which VAR is bound. Consider the following example:

```
←(STKSCAN 'X)
[STACKP]#152734/FACTORIAL
←(STKSCAN 'X 10)
[STACKP]#152731/FACTORIAL
←(STKSCAN 'X 8)
[STACKP]#152730/FACTORIAL
```

which shows that the variable X is bound in several frames on the stack as one would expect from a recursive function.

The stack appears as (using BTV)

```
COND
     X 0
FACTORIAL
COND
     X 1
FACTORIAL
COND
     X 2
FACTORIAL
COND
     X 3
FACTORIAL
COND
     X 4
FACTORIAL
EVAL
     LISPXVALUE NIL
     LISPXLINE NIL
     LISPXLISTFLG T
     HELPFLG T
     (LISPXHIST ((&) _)
     HELPCLOCK 10641
*PROG*LAM
     LISPXID ←
LISPX
ERRORSET
EVALQT
```

We will use this stack formation in examining the effect of several other stack functions in the following sections.

30.5 STACK FRAME OPERATIONS

INTERLISP provides considerable flexibility for manipulating the contents of stack frames. The caveat remains: Let the user beware! You may do your envi-

ronment grievous damage by manipulating the contents of stack frames from within your program.

30.5.1 Distinguishing Real from Dummy Frames

INTERLISP-10 places dummy frames on the stack as placeholders for certain types of functions including *PROG*LAM, *ENV*, and some block compiler entities. You may test if a frame is a dummy frame using **DUMMYFRAMEP**, which takes the form

```
Function:       DUMMYFRAMEP
# Arguments:    1
Argument:       1) A stack frame pointer, POSITION
Value:          T, if the frame is a dummy frame.
```

DUMMYFRAMEP determines if there is a link to the frame at POS, e.g., if your program contains a function call to the function that the frame represents. The types of frames mentioned above merely serve to establish environmental information and are never explicitly executed. Dummy frames are used primarily during interpretation and disappear when functions are compiled.

Consider the stack described above:

```
←(DUMMYFRAMEP '*PROG*LAM)
T
```

because *PROG*LAM is placed on the stack as a marker by INTERLISP-10, but

```
←(DUMMYFRAMEP 'FACTORIAL)
NIL
←(DUMMYFRAMEP 'COND)
T
```

Testing for a Real Frame

An alternative function, **REALFRAMEP**, tests whether a stack frame is a real frame, i.e., not a dummy frame. It takes the form

```
Function:       REALFRAMEP
# Arguments:    2
Arguments:      1) a stack pointer, POSITION
                2) an interpretation flag, INTERPFLG
Value:          POSITION, if the frame does not disappear
                when the function is compiled.
```

Consider the following example:

```
←(STKNAME 32)
COND
←(REALFRAMEP 6 T)
6
←(REALFRAMEP 6)
NIL
```

because, when the function is compiled, the stack frame representing COND disappears. INTERPFLG specifies whether the determination should be made in interpreted or compiled mode.

Note: Neither of these functions is available in INTERLISP-D because of the different mechanisms used for representing and manipulating the stack.

30.5.2 Finding a Real Stack Frame

Given that dummy and real stack frames may be intermixed on the stack, during debugging you may want to find a real stack frame that is linked somewhere below your present position. **REALSTKNTH** allows you to skip back N frames from the current position. It takes the form

```
Function:       REALSTKNTH
# Arguments:    4
Arguments:      1) a frame count, N
                2) a stack pointer, POSITION
                3) an interpretation flag, INTERPFLG
                4) an old stack pointer, OLDPOSITION
Value:          A stack pointer to the Nth frame.
```

REALSTKNTH operates like STKNTH, including taking the same type of arguments, except that it applies REALFRAMEP to each frame that it examines on the stack. Only those frames that satisfy REALFRAMEP with INTERPFLG are counted against N. If N is negative, the control chain from POSITION is followed. If N is negative, the access chain is followed.

30.5.3 Scanning Frames for Atom Bindings

An atom may be rebound several times when a program is executed. As each function is invoked and the atom passed as an argument or the atom name is reused as a local variable within a function, the atom's name will be rebound on the stack in the frame corresponding to that function invocation. You may search the stack for a particular atom binding using **FRAMESCAN**, which takes the form

```
Function:       FRAMESCAN

# Arguments:    2

Arguments:      1) an atom name, ATM
                2) a stack pointer, POSITION

Value:          The relative position of the binding of
                ATM in the basic frame of POSITION.
```

Consider the following example:

```
←(FRAMESCAN 'X 'FACTORIAL)
1

←(FRAMESCAN 'HELPFLAG '*PROG*LAM)
3
```

FRAMESCAN returns NIL if the variable is not bound in the frame specified by POSITION.

30.6 EVALUATION IN OTHER FRAMES

The purpose of a stack pointer is to allow you to preserve an evaluation environment for later usage and inspection. Each stack frame contains bindings for variables that provide a context for evaluation. INTERLISP provides a set of functions that allows you to evaluate functions in stack frames (i.e., contexts) other than the current active frame. This feature is extremely useful during debugging. It should be utilized only by experienced INTERLISP programmers.

30.6.1 Evaluation in Other Contexts

ENVEVAL allows you to evaluate an expression in another environment. It takes the form

```
Function:       ENVEVAL

# Arguments:    5

Arguments:      1) an expression, EXPRESSION
                2) an access chain pointer, APOSITION
                3) a control chain pointer, CPOSITION
                4) an ALINK release flag, AFLAG
                5) a CLINK release flag, CFLAG

Value:          The value of the expression when evaluated
                in the new context.
```

ENVEVAL creates a new active frame on the stack for EXPRESSION using the frame specified by APOSITION as its ALINK and the frame specified by

CPOSITION as its CLINK. Note that APOSITION and CPOSITION specify a sequence of frames that will describe the bindings for variables within EXPRESSION. AFLAG and CFLAG, if non-NIL, specify that APOSITION and CPOSITION, respectively, will be released after EXPRESSION is evaluated.

Applying a Function in an Environment

An alternative form, **ENVAPPLY**, applies a specified function to the given arguments within the new context. It takes the form

```
Function:       ENVAPPLY
# Arguments:    6
Arguments:      1) a function name, FN
                2) an argument list, ARGS
                3) an access chain pointer, APOSITION
                4) a control chain pointer, CPOSITION
                5) an ALINK release flag, AFLAG
                6) a CLINK release flag, CFLAG
Value:          The value returned by FN when applied to
                ARGS in the specified context.
```

30.6.2 Evaluating Expressions in an Access Environment

You may evaluate expressions in an access environment using the functions **STKEVAL** and **STKLAPPLY**. STKEVAL takes the form

```
Function:       STKEVAL
# Arguments:    3
Arguments:      1) a stack descriptor, POSITION
                2) an expression, EXPRESSION
                3) a release flag, FLAG
Value:          The value of EXPRESSION when evaluated in
                the specified context.
```

STKEVAL may be defined as

```
(DEFINEQ
     (stkeval (position expression flag)
          (ENVEVAL expression position NIL flag)
))
```

An alternative form, STKAPPLY, operates in a manner similar to STKEVAL, but applies a function to some arguments. It takes the form

```
Function:       STKAPPLY

# Arguments:    4

Arguments:      1) a stack descriptor, POSITION
                2) a function name, FN
                3) an argument list, ARGS
                4) a release flag, FLAG

Value:          The value returned by the function when
                applied to the arguments in the specified
                context.
```

STKAPPLY may be defined as

```
(DEFINEQ
     (stkapply (position fn args flag)
          (ENVAPPLY fn args position NIL flag)
))
```

In both functions, if FLAG is non-NIL, the stack descriptor will be released after the evaluation is performed.

30.7 MANIPULATING STACK POINTERS

A *stack pointer* is the address of a basic stack frame in the stack. Stack pointers are used by these functions to reference the basic stack unit, the frame. Stack pointers represent a unique data type because they are allocated from a special memory pool. INTERLISP provides a number of functions for manipulating these pointers.

30.7.1 Testing a Stack Pointer

You may test the value of an atom to determine if it is a stack pointer using **STACKP**, which takes the form

```
Function:       STACKP

# Arguments:    1

Argument:       1) a stack pointer, POSITION

Value:          POSITION, if it is a stack pointer;
                otherwise, NIL.
```

Consider the following example:

```
:(SETQ XSTKP (STKPOS (STKNAME 21)))
[STACKP]#152632/FACTORIAL
```

```
:(STACKP XSTKP)
T
```

30.7.2 Releasing a Stack Pointer

You may release a stack pointer using the function **RELSTK**. When you do so, the stack frame associated with the pointer is allocated, so you may no longer reference it. It takes the form

```
Function:       RELSTK
                RELSTKP
# Arguments:    1
Argument:       1) a stack pointer, POSITION
Value:          POSITION, but the stack frame is released.
```

If POS is not a stack pointer, RELSTK does nothing.

You may test if a stack pointer has been released using **RELSTKP**. It takes the same argument as RELSTK and returns POSITION if the stack frame corresponding to POSITION has been released.

30.7.3 Clearing an Active Stack

CLEARSTK releases all active stack pointers. It takes the form

```
Function:       CLEARSTK
# Arguments:    1
Argument:       1) a mode flag, FLAG
Value:          A list of active stack pointers or NIL.
```

If FLAG is NIL, CLEARSTK releases all active stack pointers and returns NIL.

If FLAG is T, CLEARSTK returns a list of all the active stack pointers, but does not release them, e.g.,

```
←(CLEARSTK T)
([STACKP]#152645/COND [STACKP]#152632/FACTORIAL)
```

CLEARSTK uses two lists, CLEARSTKLST and NOCLEARSTKLST, to determine what action to take for each stack pointer. CLEARSTKLST is a global variable used by EVALQT. Whenever EVALQT is called, it checks CLEARSTKLST, which may take three values:

1. If its value is T, all active stack pointers are released using CLEARSTK.
2. If its value is NIL, nothing is released.
3. If its value is a list, only the valid stack pointers on that list are released.

EVALQT is entered when INTERLISP is started up, following errors, or when the user has performed a CTRL-D or (RESET).

NOCLEARSTKLST is a global variable used by EVALQT to mediate the behavior caused by CLEARSTKLST. It is initially NIL. However, you may assign as its value a list of one or more valid stack pointers which will not be released when CLEARSTK is called because CLEARSTKLST has the value T. Generally, you will want to save a few stack pointers from being released, so it is easier to set the value of NOCLEARSTKLST than it is for CLEARSTKLST.

30.7.4 Copying Stack Frames

The stack represents an environment for the execution of a set of functions. You may save that environment by copying the appropriate stack frames using **COPYSTK**, which takes the form

```
Function:       COPYSTK
# Arguments:    2
Arguments:      1) a stack pointer, OLDPOS
                2) a stack pointer, NEWPOS
Value:          A stack pointer which is the "new" NEWPOS.
```

COPYSTK copies the specified stack frames from the frame identified by OLDPOS to the frame identified by NEWPOS. Both basic frames and the frame extensions are copied by following the access chain from OLDPOS to NEWPOS inclusive. OLDPOS must be in the access chain from NEWPOS, e.g., it must appear before NEWPOS on the stack and its function must have been called prior to the function given by NEWPOS. COPYSTK copies the stack frames at the end of the stack and returns the stack pointer of the last frame so copied.

This function is available only in INTERLISP-10.

30.8 EXITING FROM A STACK FRAME

When an error occurs, you usually will enter the Break Package. Within a break, you may use the Break Package commands, which are implemented by using the stack functions described in this chapter, or the stack functions themselves to analyze the symptoms and causes of the break. The Break Package allows you to interactively debug your program to determine what caused the error. But, more importantly, it allows you correct the error (in many cases) and continue the computation.

In some cases, you will determine the cause and want to exit the break with some value being returned to the calling function that invoked the function that was broken. Two cases come immediately to mind:

1. You are not able to repair the error because it requires extensive modification or new programming. Rather, to continue testing, you want to generate the value that the function would have returned, and continue with the computation.
2. The function that was broken was a stub that has not yet been coded. You merely want to return a value that it might have returned and continue testing at a higher level of your program architecture.

The functions described in this section allow you to return from or to a specified stack frame with a value.

Returning from a Stack Frame

RETFROM returns a value from a specified stack frame. It takes the form

```
Function:       RETFROM
# Arguments:    3
Arguments:      1) a stack frame, POSITION
                2) a value to be returned, VALUE
                3) a frame release flag, FLAG
Value:          The value of VALUE.
```

RETFROM returns from the stack frame given by POSITION with VALUE as its value. You may return from the current stack frame with the value that it might have computed by executing the expression

```
(RETFROM -1 ⟨value⟩)
```

If FLAG is not NIL and POSITION is a stack pointer, then the stack frame is released as if you had returned normally from the function represented by the stack frame.

You cannot return from the top level. Any attempt to do so will cause an error:

```
←(RETFROM T)
ILLEGAL STACK ARG
T
```

Returning to a Stack Frame

You may return to a stack frame by executing **RETTO**. Usually, you will want to perform this operation when intervening frames have compounded the error but

it was not detected until several more functions had been invoked. The basic idea is that there is a point in the stack from which the computation may continue without error provided the proper value is returned from lower-level functions. However, if you try to return a value from the broken function, it will not be utilized correctly because of the compound error problem. RETTO allows you to return to a stack frame with the proper value such that the computation may continue unimpeded. It takes the form

```
Function:       RETTO
# Arguments:    3
Arguments:      1) a stack frame, POSITION
                2) a value to be returned, VALUE
                3) a frame release flag, FLAG
Value:          The value of VALUE.
```

Returning from a Stack Frame with Evaluation

RETFROM returns the value given as its second argument. In many cases, you do not know what the value should be, but you do know how it can be computed. **RETEVAL** allows you to compute a value to be returned from a stack frame. It takes the form

```
Function:       RETEVAL
# Arguments:    3
Arguments:      1) a stack frame, POSITION
                2) an expression, EXPRESSION
                3) a frame release flag, FLAG
Value:          The value of EXPRESSION.
```

EXPRESSION is evaluated in the access environment specified by POSITION. RETEVAL returns from the stack frame specified by POSITION with the corresponding value of EXPRESSION. In most cases, POSITION will be different from the stack frame in which the computation was broken because the correct value to be returned may be computed at a higher level in the stack.

EXPRESSION may be any general S-expression. If the value to be returned is the result of evaluating a function with a known set of arguments, then you may use an alternative function, **RETAPPLY**. It takes the form

```
Function:       RETAPPLY
# Arguments:    4
Arguments:      1) a stack frame, POSITION
                2) a function, FN
```

```
                3) a list of arguments, ARGS
                4) a frame release flag, FLAG
Value:          The result of applying FN to ARGS.
```

RETAPPLY operates in a manner similar to RETEVAL.

30.9 OPERATING ON THE STACK

INTERLISP provides several functions for operating on the entire stack. Usually, you will want to search the stack for a specific frame or display the stack structure in different formats. These functions are described in the following sections.

30.9.1 Mapping Down the Stack

You may apply a function to every frame on the stack using **MAPDL** because the stack is treated as a single large list. MAPDL takes the form

```
Function:       MAPDL
# Arguments:    2
Arguments:      1) a mapping function, MAPDLFN
                2) an initial stack descriptor, MAPDLPOS
Value:          NIL.
```

Starting at MAPDLPOS, MAPDLFN is applied to each frame of the stack until the top of the stack is reached. MAPDLFN is a function of two arguments: the function name and the frame pointer itself. Consider the following example,

```
←(MAPDL (FUNCTION
                (LAMBDA (X POS)
                     (PRINT X))))
EVAL
*PROG*LAM
LISPX
ERRORSET
EVALQT
NIL
←(FACTORIAL 3)
(FACTORIAL AFTER ZEROP broken)
:(MAPDL (FUNCTION
          (LAMBDA (X POS)
               (PRIN1 POS)
```

```
                (SPACES 3)
                (PRINT (VARIABLES POS)))))
[STACKP]#152567/EVAL    (NIL NIL)
[STACKP]#152567/*PROG*LAM (HELPCLOCK LISPXHIST HELPFLAG
NIL LISPXLISTFLG LISPXLINE NIL NIL LISPXVALUE NIL NIL)
[STACKP]#152567/LISPX   (NIL LISPXID NIL NIL NIL)
...
[STACKP]#152567/BREAK1   (BRKEXP BRKWHEN BRKFN BRKCOMS
BRKTYPE NIL)
[STACKP]#152567/COND    (NIL)
[STACKP]#152567/FACTORIAL (X)
...
```

30.9.2 Searching Down the Stack

SEARCHPDL allows you to search the stack while applying a function, SEARCHPDLFN, until a stack frame yields T when SEARCHPDLFN is applied to it. It takes the form

```
Function:       SEARCHPDLFN

# Arguments:    2

Arguments:      1) a search function, SEARCHPDLFN
                2) an initial stack position, SEARCHPDLPOS

Value:          (⟨name⟩ . ⟨frame⟩); otherwise, NIL.
```

SEARCHPDL operates in a manner similar to MAPDL. Starting at the stack frame specified by SEARCHPDLPOS, it applies SEARCHPDLFN to every frame. If the search function returns T, then SEARCHPDL halts and returns a value consisting of (⟨name⟩ . ⟨frame⟩) where ⟨name⟩ is the name of the function represented by the frame, and ⟨frame⟩ is the associated address.

SEARCHPDLFN is a function of two arguments: the name of the function represented by the frame, and the frame itself.

The INTERLISP Compiler

INTERLISP, like most LISP systems, is usually thought of as an interpreted language. However, INTERLISP provides a compiler that produces efficient code. This chapter discusses many of the features of the compiler but avoids discussion of the assembler (available only in INTERLISP-10/VAX/370) or other machine-dependent features. We will, however, show some samples of compiled code.

The Compiler may be used to compile functions or files that contain one or more function definitions. Compiled code may be written into memory for later execution or to a file for later loading.

Note: We do not discuss the use of Block Compiling in this text because we do not feel it is required on the newer versions of LISP machines that are appearing in the commercial marketplace.

31.1 THE COMPILER DIALOGUE

The Compiler will ask you a set of questions that help it to determine the actions to be taken when compiling. Your answers to these questions are used to set several variables which are used by the various Compiler functions and other system packages that it calls.

The first question that the Compiler asks you concerns the generation of a listing. It appears as:

```
LISTING?
```

You may respond to this question in several ways. There are actually two sets of answers. The first set includes

```
1          Print the output of Pass 1, the INTERLISP
           macrocode
```

2	Print the output of Pass 2, the machine language code
YES	Print the output of both passes
NO	Print no listings

The free variable LAPFLG is set to the answer you provide. If you answer 1, 2, or YES, the Compiler will ask you about a file name as follows:

```
FILE:
```

You should respond with the name of a file to which you want the output to be written. The free variable LSTFIL is set to the name of the file.

Next, the Compiler will ask you about redefining your functions via

```
REDEFINE?
```

If you answer YES, each function will be redefined as it is compiled. That is, its current definition becomes the compiled code. If you answer NO, the current function definition remains unchanged. The free variable STRF is set to your response.

Next, the Compiler asks you about saving source definitions. If you answered YES to the redefinition question, the Compiler will ask you

```
SAVE EXPRS?
```

If you answer YES to this question, the definitions of any functions that are EXPRs will be saved on their property lists under the property EXPR. Otherwise, the source definitions are discarded. The free variable SVFLG is set to your answer.

Finally, the Compiler will ask you for the name of an output file via

```
OUTPUT FILE?
```

If you want to save your compiled functions for later loading, you should answer YES to this question. If you answer T or TTY:, the output will be typed at your terminal (not recommended as it may be rather lengthy). If you answer NO, your compiled definitions will not be saved (except in memory).

If you answered YES, the Compiler will ask you for a file name to which the output should be written. If the file is already open, newly compiled code will be appended to the end of the file. The free variable LCFIL is set to the name of the file.

I mentioned above that there was a second set of answers to the question concerning listings. These answers, described below, specify a complete mode of operation for the Compiler.

You may respond to the listing question with these answers as well:

S	Same as the last setting (which is retained by the Compiler)
F	Compile to a file; functions are not redefined
ST	Store new definitions for functions and save their current definitions under their EXPR properties
STF	Store new definitions, but forget their EXPR definitions

31.2 COMPILATION ISSUES

When compiling functions, there are a number of issues that you must be aware of that will cause functions to operate differently when compiled. These issues are discussed in this section.

31.2.1 Compiling NLAMBDA Functions

The Compiler must prepare the arguments to a function in one of three ways:

1. Evaluated (SUBR, SUBR*, EXPR, EXPR*, CEXPR, CEXPR*)
2. Unevaluated, spread (FSUBR, FEXPR, CFEXPR)
3. Unevaluated, nospread (FSUBR*, FEXPR*, CFEXPR*)

The Compiler derives its information about how to prepare a function call from several sources. These include (in descending order)

1. The function definition in the file
2. The lists NLAMA, NLAML, or LAMS
3. The function definition in memory which is assumed to be the desired type

The lists NLAMA, NLAML, and LAMS should contain the names of functions which are NLAMBDA nospread, NLAMBDA spread, and LAMBDA functions, respectively. These lists may be used to override the function type as stored in its definition if the function cannot be found in the file that is being compiled.

If the function is not contained within a file or on one of the lists mentioned above, or is currently defined within your environment, the Compiler calls COMPILEUSERFN. COMPILEUSERFN has both a value and a function definition.

If the value of COMPILEUSERFN is not NIL, the Compiler applies the value of COMPILEUSERFN to the expression that it attempted to compile and the CDR of the expression. That is, the Compiler performs

```
(APPLY* COMPILEUSERFN (CDR ⟨expression⟩) ⟨expression⟩)
```

If a non-NIL value is returned, it is compiled instead of the expression. If NIL is returned, the Compiler compiles the expression as a LAMBDA spread function which has not yet been defined.

COMPILEUSERFN is only called when the Compiler encounters an expression whose CAR is not a known or defined function name.

CLISP uses COMPILEUSERFN to instruct the Compiler about how to compile iterative statements, IF-THEN-ELSE statements, and other constructs.

If the Compiler cannot determine the function type by any of the means above, it assumes the function to be a LAMBDA spread function, and adds its name to the list ALAMS (for Assumed LAMbdas). ALAMS is not used by the Compiler, but is maintained for your benefit so that you may determine whether the Compiler made any incorrect assumptions or not.

31.2.2 Declarations

A *declaration* is indicated by a DECLARE: expression. Declarations are included in functions and in files to specify what actions should be taken when the function or file is compiled. A declaration takes the form

```
(DECLARE: . [⟨filepkgcoms⟩ or ⟨flags⟩])
```

Normally, expressions included in a symbolic file are treated as follows:

1. They are evaluated when loaded.
2. They are copied to a compiled file when the symbolic file is compiled.
3. They are not evaluated at compile time.

A declaration allows you to override these actions. DECLARE: is defined as an NLAMBDA nospread function. It evaluates each of its arguments in turn, although this may be modified by the value of the arguments themselves. The following tags may appear in a declaration:

EVAL@COMPILE	Succeeding expressions are evaluated at compilation.
EVAL@LOAD	Evaluate succeeding expressions when the file is loaded.
DOCOPY	When compiling, copy the following forms into the compiled file.

EVAL@COMPILEWHEN	Evaluate the succeeding expressions if the condition following is true.
EVAL@LOADWHEN	Evaluate the following expressions based on the value of the first expression immediately after the tag. If the tag is T, all expressions are evaluated at loading.
FIRST	For expressions that are to be copied to a compiled file, this tag specifies that the following expressions are to appear at the beginning of the file.

DECLARE: expressions are processed in a special manner by the Compiler. DECLARE: is used in two ways:

1. To specify expressions that are to be evaluated at compile time, presumably to affect the compilation (for example, using FMEMB instead of MEMB).
2. To indicate which expressions appearing in the symbolic file are not to be copied or compiled into the output file.

31.2.3 Open Functions

When a function is called from a compiled function, the stack frame must be initialized with the argument lists and control information to ensure proper linkage to and return from the function. If your functions are relatively small (a few expressions), then the setup time may exceed the actual execution time of the function. When your system consists of many small functions (as encouraged by a structured programming effort), the housekeeping time may dominate the actual execution time. Many small system functions are compiled *open*; that is, they do not require a function call, but are translated into in-line code. Other system functions are compiled open because they are frequently used. The IRM describes the list of functions which are always compiled open. You may force either your own or other system functions to compile open through appropriate use of MACROs.

31.2.4 Constants

Constants are expressions which define "constant," i.e., immutable values. In most high-level languages, when constants are compiled, they refer to a single location in memory. To achieve this goal in INTERLISP, the function CONSTANTS is provided. It takes the form

```
Function:       CONSTANTS
# Arguments:    1-N
Arguments:      1-N) a list of variables, VAR[1] ...
                     VAR[n]
Value:          A list of the variables declared as
                constants.
```

CONSTANTS defines VAR[1] ... VAR[n] to be compile-time constants. Whenever the Compiler encounters a free reference to a constant, it will compile the expression (CONSTANT ⟨variable⟩) instead.

The function CONSTANT allows you to define certain expressions as descriptions of a constant value. For example, you may define a scratch list of length 20 as follows:

```
←(CONSTANT (TO 20 COLLECT NIL))
(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL NIL NIL NIL NIL)
```

When encountered during interpretation, the expression will be evaluated each time it is encountered. However, during compilation, it is evaluated exactly once.

31.2.5 COMPILETYPELST

The Compiler understands how to compile expressions and variables. You may instruct the Compiler in the handling of certain expressions and variables via macros, declarations, COMPILEUSERFN, and COMPILETYPELST.

COMPILETYPELST is a list of entries of the form

```
(⟨typename⟩ . ⟨function⟩)
```

It instructs the Compiler about what to do when it encounters a datatype other than a list or an atom. Thus, it has a close correspondence to the function DEFEVAL. Whenever the compiler encounters a datum whose type is neither list nor atom, the Compiler looks up the type name on COMPILETYPELST. If an entry appears on this list whose CAR is the type name, the Compiler applies the expression which is the CDR of the entry to the datum. The expression should return a non-NIL value which will be compiled instead. If it returns NIL or there is no entry for the given type name on COMPILETYPELST, the Compiler merely compiles (QUOTE ⟨datum⟩).

31.2.6 Compiling CLISP

The Compiler does not know about CLISP. However, it is often asked to compile expressions or functions that contain CLISP constructs. To compile these ex-

pressions, the expressions must be DWIMIFYed You may assist the Compiler in performing this translation in several ways:

1. If DWIMIFYCOMPILEFLG is T, the Compiler always DWIMIFYs expressions before compiling them. It is initially NIL.
2. If a file has the property FILETYPE with the value CLISP on its property list, then the Compiler functions assume that DWIMIFYCOMPILEFLG is T for the file, even if it is NIL.
3. If the function definition has a CLISP: declaration, including a null declaration of the form (CLISP:), the function will automatically be DWIMIFYed before compiling.

COMPILEUSERFN does this automatically for all CLISP statements and record package statements, so if these are the only statements in the function, you do not have to do anything further.

31.3 COMPILER FUNCTIONS

The Compiler allows you to compile either functions or files containing functions. When you compile functions, you should be aware of certain actions taken by the Compiler:

1. If the function has been modified by BREAKing, TRACEing, or ADVISEing, it is restored to its original state before compilation. The Compiler notifies you with a message of the form "⟨function⟩ UNBROKEN".
2. If the function is not defined as an EXPR, the Compiler searches its property list for the property EXPR; if found, it uses its value as the definition of the function.
3. The Compiler prints a warning

   ```
   (⟨function⟩ NOT COMPILEABLE)
   ```

   and proceeds to the next function in the list.

31.3.1 Compiling Functions

To compile a function, you invoke the function **COMPILE**, which takes the form

```
Function:       COMPILE
# Arguments:    2
Arguments:      1) a function list, FNS
                2) a destination flag, FLG
Value:          The value of FNS.
```

FNS is a function or a list of functions to be compiled. COMPILE asks a set of standard questions. It then compiles each function in FNS using its current definition. If FLG is T, and compiled code is to being written to a file, the file is closed when COMPILE terminates. If FLG is NIL, successive calls to COMPILE will append additional compiled code to the end of the file. Consider the following example:

```
←(COMPILE 'FACTORIAL)
listing? Yes
file: F.LST
redefine? No
output file? No
(FACTORIAL (X) NIL)
(FACTORIAL)
←(PRINTDEF (READFILE 'F.LST))
((FACTORIAL (X)
     NIL)
     (ENTERF 1 0 0 0)
     -1
     (LDV X 0)
     (SKES 0)
     (JR 2)
     (LQI 1)
     (RET)
     2
     (PIUNBX (VREF X 0))
     (LPUSHN 1)
     (PIUNBX (VREF X 0))
     (SUBI 1 , 1)
     (FCLL MKN)
     (ACLL FACTORIAL 1)
     (FCLL IUNBOX)
     (IMULM 1 , 0 (CP)
     (LPOPN 1)
     (FCLL MKN)
     (RET)
     FACTORIAL
     (JSP 7 , ENTERF)
     (262144 0)
     (0 PLITORG)
     -1
     (HRRZ 1 , (VREF X 0))
     (CAIE 1 , ASZ 0)
     (JRST (TREF 2))
```

```
(HRRZI 1 , ASZ 1)
(POPJ CP ,)
2
(PIUNBX (VREF X 0))
(PUSH CP, = 4456449)
(PUSH CP , 1)
(PIUNBX (VREF X 0))
(SUBI 1 , 1)
(PUSHJ CP , MKN)
(ACCALL 1 , ' FACTORIAL)
(PUSHJ CP , IUNBX)
(IMULM 1 , 0 (CP))
(POP CP , 1)
(SUB CP , BHC 1)
(PUSHJ CP , MKN)
(POPJ CP ,)
LITORG 4456449 PLITORG (VARIABLE-VALUE-CELL X . 11)
FACTORIAL)
```

31.3.2 Compiling a Definition

COMPILE1 compiles a definition for a function. It takes the form

```
Function:       COMPILE1

# Arguments:    2

Arguments:      1) a function name, FN
                2) a definition, DEFINITION

Value:          The compiled form of the definition.
```

Consider the following example:

```
←(COMPILE1 'ADD3 '(LAMBDA (X Y Z) (IPLUS X Y Z)))
(ADD3 (X Y Z) NIL)
(ADD3 (X Y Z) NIL
     (ENTERF 3 0 0 0)
-1   (PIUNBX (VREF X 0))
     (LPUSHN 1)
     ...
     ⟨rest of the definition⟩
```

COMPILE1 is used by most of the other Compiler functions. It redefines FN with the compiled code of DEFINITION if STRF has the value T. If DEFINITION contains CLISP operators and DWIMIFYCOMPFLG is T, COMPILE1 calls DWIMIFY before compiling DEFINITION.

31.3.3 Compiling Symbolic Files

TCOMPL is used to compile symbolic files. It produces a file containing compiled code that is equivalent to the S-expressions in the original file. It takes the form

```
Function:       TCOMPL
# Arguments:    1
Argument:       1) a list of files, FILES
Value:          A list of the output files corresponding
                to the compiled files.
```

FILES is either an atom, e.g., the name of a file to be compiled, or a list of files to be compiled. TCOMPL asks the standard Compiler questions (see Section 31.1) except for the one concerning the output file. The compiled code will be written to a file of the same name as the source file, but whose suffix is either COM for INTERLISP-10 or DCOM for INTERLISP-D.

TCOMPL will not compile any functions whose name appears on the list which is the value of DONTCOMPILEFNS. This variable initially has the value NIL.

If FILES is a list, TCOMPL processes the list one file at a time. It operates as follows:

1. The entire file is read into memory.
2. For each FILECREATED expression, it notes the functions that were marked as changed by the File Package.
3. If the file name atom has the property FILETYPE with value CLISP or a list containing the value CLISP, TCOMPL sets DWIMIFYCOMPFLG to T so that any expressions in the file are DWIMIFYed before being compiled.
4. For each DEFINEQ, TCOMPL will

   add NLAMBDA names to the list NLAMA or NLAML
   add LAMBDA names to the list LAMS

   so that calls to these functions are properly generated.
5. Expressions beginning with the atom DECLARE: are processed as described in Section 31.2.2.
6. All other expressions are collected for writing to the output file.
7. Each function is compiled, and the compiled code is written to the output file.
8. All other expressions are written to the output file.
9. The file is closed.

INTERLISP/370 Convention

Because of the limited name space available under VM/SP, INTERLISP/370 uses a different naming convention. All files in INTERLISP/370 consist of five characters followed by a # followed by two digits indicating the current version. The file type is always LISP. INTERLISP/370 creates a file name for the compiled code by appending the first four characters of the file name to the letter X. Thus, if I compiled a file whose name is MAPFN#00, INTERLISP/370 would create a file named XMAPF#00 into which it writes the compiled code.

Consider the following example:

```
←(TCOMPL 'COMPLEX)
listing? Yes
file? COMPLEX.LST
redefine? No
(COMPLEX (R I) NIL)
(REAL (CX) NIL)
(IMAG (CX) NIL)
(CPLUS (CX1 CX2) NIL
...
and so on for other functions in the file COMPLEX.
(〈KAISLER〉COMPLEX.COM.1)
```

31.3.4 Recompiling a File

When you make changes to a file, you often want to recompile those functions. If your files contain many functions, recompiling the entire file may be very time-consuming. **RECOMPILE** allows you to update a compiled file without having to recompile every function in the file. RECOMPILE does this by compiling the changed functions. It then copies the definitions of unchanged functions from the compiled file to a new file along with the compiled definitions of changed functions. It takes the form

```
Function:       RECOMPILE

# Arguments:    3

Arguments:      1) a prettyfile, PFILE
                2) a compiled code file, CFILE
                3) a list of functions, FNS

Value:          The name of the new compiled file.
```

PFILE is the name of a source file in PRETTYDEF format. CFILE is the name of a compiled code file containing definitions that may be copied. FNS indicates which functions in PFILE are to be recompiled. PFILE, not FNS, drives the action of RECOMPILE.

RECOMPILE will ask you the standard compiler questions, except for "output file?". It uses the file map to locate the definitions of functions which must be recompiled. For each function in the file, RECOMPILE determines whether it must be compiled or not. If so, it is compiled and the code written to the next version of CFILE. If not, the corresponding compiled code is merely copied from the old version of CFILE to the new one. As it goes, RECOMPILE builds a file map for the new version of CFILE.

31.4 COMPILED CODE

In general, it is not very interesting to examine the compiled code produced by INTERLISP because of all the facilities available to you to examine your programs at the source language level. However, some readers may be curious to see what compiled code looks like. For your edification, I duplicate some examples of compiled code for a few functions that have been mentioned in previous examples. The machine language is that produced by INTERLISP-D running on a Xerox 1100 Scientific Information Processor.

In the following examples, I show the INTERLISP source code followed by the compiled code that was printed in the listing file.

First, a very simple function:

```
(DEFINEQ
     (alldemons? (node)
          (GETPROP node 'demons)
     ))

ALLDEMONS? (node)   LAMBDA
#⟨node⟩ 'DEMONS [GETP] RETURN

ALLDEMONS?
name table:
(L (0 NODE))
code length:   argtype: 0
(IVAR 0)
ACONST 0
(ATOM . DEMONS)
GETP
RETURN
```

Next, a slightly more complex function:

```
(DEFINEQ
     (do-off-of-demon (off)
          (COND
               ((EQUAL (CAR off) (CAR inheritance))
                    (MAPCAR (GETPROP node 'offspring)
```

```
                                (FUNCTION do-demon))))
    ))
```

Note that INHERITANCE and NODE are both used freely by this function.

```
    DO-OFF-OF-DEMON (node) LAMBDA
    ⟨OFF⟩ [CAR] ⟨INHERITANCE⟩ [CAR] [EQUAL] NFJUMP 12
    BIND[$w,$z,$v] ⟨OFF⟩ JUMP 7
4:  COPY [CAR] [DO-DEMON] SETQ ⟨$w⟩ POP #⟨$z⟩ NTJUMP 10
    #⟨$w⟩ [CONS1] SETQ ⟨$v⟩ JUMP 11
10: #⟨$w⟩ [\RPLCONS]
11: SETQ ⟨$z⟩ POP
7:  [CDR] [LISTP] NTJUMP 4
    #⟨$v⟩
12: RETURN
```

```
    DO-OFF-OF-DEMON
    name table:
    (I 0 OFF 3 INHERITANCE)
    code length:   argtype: 0
    (IVAR 0) CAR
    (FVAR 3) CAR
    FN2 0
    (FN.EQUAL)
    NFJUMPX 36Q
    BIND 60Q 2
    (IVAR 0) JUMPX 23Q
    COPY CAR FN1 0 (FN.DO-DEMON)
    (PVAR← ↑ 0)
    (PVAR 1) NTJUMPX 7
    (PVAR 0)
    'NIL CONS
    (PVAR← 2)
    (JUMP 1)
    (PVAR 0)
    RPLCONS
    (PVAR← ↑ 1)
    CDR LISTP NTJUMPX 355Q (PVAR 2) RETURN
```

31.5 COMPILER ERROR MESSAGES

The Compiler may emit a large number of error messages in response to unusual conditions it encounters while trying to compile your source files. These messages are listed below with a statement of the possible cause.

⟨fn⟩ NOT COMPILEABL
An EXPR definition for FN cannot be found by the Compiler. No code is produced for the function. The Compiler proceeds to the next function, if any.

⟨fn⟩ NOT FOUND
RECOMPILE has attempted to copy a compiled code definition for FN from CFILE and has not been able to find one. No code is copied and the Compiler proceeds to the next function, if any.

(* --) COMMENT USED FOR VALUE
A comment appears in a context where its value will be used as an argument to a function. The compiled function will run, but the value at the point where the comment appeared will be "undefined".

(⟨form⟩)—NON-ATOMIC CAR OF FORM
You intended to use the value of ⟨form⟩ as a function name. The Compiler assumes this and compiles the expression as if APPLY* had been used.

(SETQ ⟨var⟩ ⟨expr⟩ --) BAD SET
You specified a SETQ of more than two arguments in your source code.

⟨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.

⟨fn⟩—NO LONGER INTERPRETED AS FUNCTIONAL ARGUMENT
The Compiler has assumed that FN is the name of a function. This message appears when FN is not defined, but is a local variable of the function being compiled. That is, you intended to APPLY the value of FN, but did not do so.

⟨fn⟩—ILLEGAL RETURN
A RETURN is encountered in the function definition which does not occur within the scope of a PROG.

⟨tag⟩—ILLEGAL GO
A GO was encountered which did not appear within the scope of a PROG.

⟨tag⟩—MULTIPLY DEFINED TAG
A PROG label has been defined more than once within the scope of the same PROG. The second definition is ignored.

⟨tag⟩—UNDEFINED TAG
A PROG label has not been defined within the scope of a PROG expression.

⟨var⟩—NOT A BINDABLE VARIABLE
VAR is either NIL, T, or not a literal atom, but one which you tried to bind somehow in an expression, PROG, or function definition.

⟨var⟩ ⟨value⟩ -- BAD PROG BINDING
This message occurs when there is an erroneous PROG binding of the form (⟨var⟩ ⟨value1⟩ ... ⟨valueN⟩).

Various other messages appear during block compiling and, for INTERLISP-10/VAX/370, during the assembly phase of the compilation process. Consult the IRM for more information on these errors.

References

[abra68]
Abrahams, P.W.,
Symbol Manipulation Languages,
contained in:
Advances in Computers, Vol. 9,
edited by F. Alt and M. Rubinoff,
Academic, New York, 1968

[agre79]
Agre, P.
Functions as Data Objects: The Implementation of Functions in LISP
Dept. of Computer Science, Univ. of MD, TR-726, 1979

[aho83]
Aho, A., J. Hopcroft, and J.D. Ullman
Data Structures and Algorithms
Addison-Wesley, Reading, MA, 1983

[aiel83]
Aiello, N.
A Comparative Study of Control Strategies for Expert Systems: AGE Implementations of Three Variations of PUFF
Proceedings, of NCAI-83, Washington, DC, 1983

[alle79a]
Allen, J.
An Overview of LISP
Byte, Vol. 4, #8, 1979

[alle79b]
Allen, J.
Anatomy of LISP
McGraw-Hill, New York, 1979

[back78]
Backus, J.
Can Programming Be Liberated from the von Neumann Style?
A Functional Style and Its Algebra of Programs
Communications of the ACM, Vol. 21, #8, 1978

[bake78a]
Baker, H.G., Jr.
List Processing in Real-time on a Serial Computer
Communications of the ACM, Vol. 21, #4, 1978

[bake78b]
Baker, H.G.
Shallow Binding in Lisp 1.5
Communications of the ACM, Vol. 21, #7, 1978

[balz73]
Balzer, R.M.
A Global View of Automatic Programming
Proceedings of 3rd IJCAI, Stanford, CA, 1973, pp. 494-499

[bars79]
Barstow, D.R.
Knowledge-Based Program Construction
North-Holland, New York, 1979

[bens81]
Benson, E. and M.L. Griss
SYSLISP: A Portable LISP-based Systems Implementation Language
Dept. of Computer Science, Univ. of Utah, UCP-81, Salt Lake City, UH, 1981

[bobr72]
Bobrow, D.G.
Requirements for Advanced Programming Systems for List Processing
Communications of the ACM, Vol. 15, #7, 1972

[bobr73a]
Bobrow, D.G. and B. Wegbreit
A Model and Stack Implementation of Multiple Environments
Communications of the ACM, Vol. 16, #10, 1973

[bobr73b]
Bobrow, D. and B. Wegbreit
A Model for Control Structures for Artificial Intelligence Programming Languages
3rd IJCAI, Stanford, CA, 1973, pp. 246-253

[bobr74]
Bobrow, D.G. and B. Raphael
New Programming Languages for Artificial Intelligence
ACM Computing Surveys, Vol. 6, #3, 1974

[bobr79]
Bobrow, D. and D. Clark
Compact Encodings of List Structures
ACM Transactions on Programming Languages and Systems, Vol. 1, #2, 1979, pp. 266-286

[bobr80]
Bobrow, D.
Managing Reentrant Structures Using Reference Counts
ACM Transactions on Programming Languages and Systems, Vol. 2, #3, 1980, pp. 269-273

[bode77]
Boden, M.
Artificial Intelligence and Natural Man
Basic Books, New York, 1977

[boye73]
Boyer, R.S. and J.S. Moore
Proving Theorems about LISP Functions
Proceedings of 3rd IJCAI, Stanford, CA, 1973, pp. 486-493

[broo82]
Brooks, R.A., R.P. Gabriel, and G.L. Steele, Jr.
An Optimizing Compiler for Lexically Scoped LISP
ACM SIGPLAN Notices, Vol. 17, #6, 1982

[burt80a]
Burton, R.R., L.M. Masinter, et al.
Interlisp-D: Overview and Status
Proceedings of the 1980 Lisp Conference
Stanford and Xerox PARC, SSL-80-4, 1980

[burt80b]
Burton, R.R.
Interlisp-D Display Facilities
Xerox Palo Alto Research Center, SSL-80-4, 1980

[char80]
Charniak, E., C.K. Riesbeck, and D.V. McDermott
Artificial Intelligence Programming
Lawrence Erlbaum Associates, Hillsdale, NJ, 1980

[clar77]
Clark, D.W. and C.C. Green
An Empirical Study of List Structure in Lisp
Communications of the ACM, Vol. 20, #2, 1977

[cohe81]
Cohen J.
Garbage Collection of Linked Data Structures
ACM Computing Surveys, Vol. 13, #3, 1981

[darl73]
Darlington, J. and R.M. Burstall
A System which Automatically Improves Programs
3rd IJCAI, Stanford, CA, 1973, pp. 479-485

[deut76]
Deutsch, L.P. and D.G. Bobrow
An Efficient Incremental, Automatic Garbage Collector
Communications of the ACM, Vol. 19, #9, 1976

[deut79]
Deutsch, L.P.
Experience with a Microprogrammed Interlisp System
IEEE Transactions on Computers, Vol. C-28, #10, 1979

[dolo78]
Dolotta, T.A., R.C. Haight, and J.R. Mashey
The Programmer's Workbench
Bell System Technical Journal, Vol. 57, #6, 1978

[elli80]
Ellis, J.R.
A LISP Shell
SIGPLAN Notices, Vol. 15, #5, 1980, pp. 24-34

[fain81]
Fain, J., et al.
The ROSIE Language Reference Manual
Rand Corporation, Santa Monica, CA, N-1647-ARPA, 1981

[fain82]
Fain, J., et al.
Programming in ROSIE: An Introduction by Means of Examples
Rand Corporation, Santa Monica, CA, N-1646-ARPA, 1982

[fire80]
Firestone, R.M.
An Experimental LISP System for the Sperry Univac 1100 Series
SIGPLAN Notices, Vol 15, #1, 1980, pp. 117-129

[fode81]
Foderaro, J.K. and K.L. Sklower
The Franz Lisp Manual
University of California at Berkeley, 1981

[forg81]
Forgy, C.L.
OPS5 Manual
Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, 1981

[fost67]
Foster, J.M.
List Processing
American Elsevier, New York, 1967

[grah79]
Graham, N.
Artificial Intelligence
TAB Books, Blue Ridge Summit, PA, 1979

[gris81]
Griss, M.L. and A.C. Hearn
A Portable LISP Compiler
Software—Practice and Experience, Vol. 11, pp. 541-605, 1981

[gris82a]
Griss, M.L.
Portable Standard LISP: A Brief Overview
Dept. of Computer Science, Univ. of Utah, USCG-58, 1982

[gris82b]
Griss, M.L., E. Benson, and A.C. Hearn
Current Status of a Portable LISP Compiler
ACM SIGPLAN Notices, Vol. 17, #6, 1982

[hase84]
Hasemer, T.
An Introduction to LISP
contained in
O'Shea, T. and M. Eisenstadt
Artificial Intelligence: Tools, Techniques, and Applications
Harper & Row, New York, 1984

[haye80]
Hayes-Roth, F.

Matching and Abstraction in Knowledge Systems
Rand Corporation, p-6440, Santa Monica, CA, 1980

[haye82]
Hayes-Roth, F., et al.
Rationale and Motivation for ROSIE
Rand Corporation, Santa Monica, CA, N-1648-ARPA, 1982

[horo82]
Horowitz, E. and S. Sahni
Fundamentals of Data Structures
Computer Science Press, Rockville, MD, 1982

[jone82]
Jones, M.A.
A Comparison of LISP Specification of Function Definition and Argument Handling
ACM SIGPLAN Notices, Vol. 17, #8, 1982

[kais80]
Kaisler, S.H.
The Design of Operating Systems for Small Computers
John Wiley, New York, 1982

[kapl80]
Kaplan, R.M., B.A. Sheil, and R.R. Burton
The Interlisp-D I/O System
Xerox Palo Alto Research Center, SSL-80-4, 1980

[kern76]
Kernighan, B.W. and P.J. Plauger
Software Tools
Addison-Wesley, Reading, MA, 1976

[korn79]
Kornfeld, W.A.
Pattern-directed Invocation Languages
Byte, Vol. 4, #8, 1979

[knut68]
Knuth, D.E.
The Art of Computer Programming: Fundamental Algorithms
Addison-Wesley, Reading, MA, 1968

[laub79]
Laubsch, J., G. Fisher, and H.D. Bocker
LISP-based Systems for Education
BYTE, Vol. 4, #8, 1979

[laub84]
Laubsch, J.
Advanced LISP Programming
contained in:
O'Shea, T. and M. Eisenstadt
Artificial Intelligence: Tools, Techniques, and Applications
Harper & Row, New York, 1984

[levi80]
Levine, J.
Why a Lisp-Based Command Language?
SIGPLAN Notices, Vol. 15, #5, 1980, pp. 49-53

[lieb80]
Lieberman, H. and C. Hewitt
A Real Time Garbage Collector That Can Recover Temporary Storage Quickly
AI Memo # 569, MIT AI Laboratory, Cambridge, MA, 1980

[lieb83]
Lieberman, H. and C. Hewitt
A Real-Time Garbage Collector Based on the Lifetimes of Objects
Communications of the ACM, Vol. 26, #6, 1983

[mart79]
Marti, J.B., et al.
Standard LISP Report
SIGPLAN Notices, Vol. 14, #10, pp. 48-68, 1979

[masi80]
Masinter, L.M. and L.P. Deutsch
Local Optimization in a Compiler for Stack-based Lisp Machines
Proceedings of the 1980 Lisp Conference
Stanford University and Xerox Palo Alto Research Center, SSL-80-4, 1980

[math77]
MathLab Group
MACSYMA Reference Manual, Version 9
MIT Laboratory for Computer Science, Cambridge, MA, 1977

[mcca72]
McCarthy, J., et al.
Lisp 1.5 Programmer's Manual, 2nd ed.
MIT Press, Cambridge, MA, 1972

[mcca78]
McCarthy, J.
History of Lisp
SIGPLAN Notices, Vol 13, #8, 1978

[moor79]
Moore, J.S., II
The Interlisp Virtual Machine Specification
Xerox Palo Alto Research Center, CSL 765, Revised 3/79

[nico81]
Nicol, R.L.
Symbolic Differentiation a' la LISP
Byte, Vol. 6, #9, 1981

[nova82]
Novak, G.S., Jr.
GLISP User's Manual
Stanford Univ., STAN-CS-82-895, Menlo Park, CA, 1982

[oste81]
Osterweil, L.
Software Environment Research: Directions for the Next Five Years
IEEE Computer, Vol. 14, #4, 1981, pp. 35-44

[prin79]
Prini, G. and M. Rudalics
The Lambdino Storage Management System
Byte, Vol. 4, #8, 1979

[rich83]
Rich, E.
Artificial Intelligence
McGraw-Hill, New York, 1983

[sace76]
Sacerdoti, E.D.
QLISP—A Language for the Interactive Development of Complex Systems
AFIPS National Computer Conference, Vol. 40, 1976

[salt80]
Salter, R., T.J. Brennan, and D.P. Friedman
CONCUR: A Language for Continuous Concurrent Processes
Computer Languages, Vol. 5, Pergamon, London, 1980, pp. 163-189

[same82]
Samet, H.
Code Optimization Considerations in List Processing Systems
IEEE Transactions on Software Engineering, Vol. SE-8, #2, 1982

[sand78]
Sandewall, E.
Programming in the Interactive Environment: The LISP Experience
ACM Computing Surveys, Vol. 10, #1, 1978

[shan80]
Shankar, K.S.
Data Structures, Types, and Abstractions
IEEE Computer, Vol. 13, #4, 1980, pp. 67-77

[shap79]
Shapiro, S.C.
Techniques of Artificial Intelligence
D. Van Nostrand Co., New York, 1979

[shne80]
Shneiderman, B.
Software Psychology
Winthrop Publishers, Cambridge, MA, 1980

[sikl76]
Siklossy, L.
Let's Talk Lisp
Prentice-Hall, Englewood Cliffs, NJ, 1976

[smit73]
Smith, D.C. and H.J. Enea
Backtracking in MLISP2: An Efficient Backtracking Method for LISP
3rd IJCAI, Stanford, CA, 1973, pp. 677-685

[sowa83]
Sowa, J.F.
Conceptual Structures: Information Processing in Mind and Machine
Addison-Wesley Systems Programming Series, Reading, MA, 1983

[stee76a]
Steele, G.L., Jr.
LAMBDA: The Ultimate Declarative
MIT AI Laboratory, AI Memo 379, Cambridge, MA, 1976

[stee76b]
Steele, G.L., Jr.
LAMBDA: The Ultimate Imperative
MIT AI Laboratory, AI Memo 353, Cambridge, MA, 1976

[stee83]
Steele, Guy L., Jr.
Common Lisp Reference Manual
Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, 1983

[stee79]
Steels, L.
Procedural Attachment
AI Memo #543, MIT AI Laboratory, Cambridge, MA, 1979

[stef82]
Stefik, M., et al.
The Organization of Expert Systems: A Prescriptive Tutorial
Xerox Palo Alto Research Center, VLSI-82-1, 1982

[stou79]
Stoutmeyer, D.R.
LISP Based Symbolic Math Systems
Byte, Vol. 4, #8, 1979

[suss81]
Sussman, G.J., et al.
Scheme-79—Lisp on a Chip
IEEE Computer, Vol. 14, #7, 1981, pp. 10-21

[symb83]
Symbolics, Inc.
3600 Technical Summary, Symbolics, Cambridge, MA, 1983

[taft79]
Taft, S.T.
The Design of an M6800 LISP Interpreter
Byte, Vol. 4, #8, 1979

[teit73]
Teitelman, W.
CLISP—Conversational LISP
3rd IJCAI, Stanford, CA, 1973, pp. 686-690

[teit78]
Teitelman, W.
Interlisp Reference Manual
Xerox Palo Alto Research Center, Palo Alto, CA, 1978

[teit81]
Teitelman, W. and L. Masinter
The Interlisp Programming Environment
IEEE Computer, Vol. 14, #4, pp. 25-34, 1981

[tesl73]
Tesler, L., H.J. Enea, and D.C. Smith
The LISP70 Pattern Matching System
3rd IJCAI, Stanford, CA, pp. 671-676

[tour84]
Touretzky, D.S.
LISP: A Gentle Introduction to Symbolic Computation
Harper & Row, New York, 1984

[trac80]
Tracton, K.
Programmer's Guide to LISP
TAB Books, Blue Ridge Summit, PA, 1980

[wate78]
Waterman, D.A. and F. Hayes-Roth, eds.
Pattern-Directed Inference Systems
Academic, New York, 1983

[wate83]
Waters, R.C.
User Format Control in a LISP Prettyprinter
ACM Transactions on Programming Languages and Systems, Vol. 5, #4, 1983, pp. 513-531

[wein81]
Weinreb, D. and D. Moon
LISP Machine Manual
Symbolics, Inc., Cambridge, MA, 1981

[wied77]
Wiederhold, G.
Database Design
McGraw-Hill, New York, 1977

[wile84]
Wilensky, R.
LISPcraft
W.W. Norton & Co., San Francisco, CA, 1984

[wins81]
Winston, P. and K.P. Horn Berthold
Lisp
Addison-Wesley, Reading, MA, 1981

Index

This index contains references to all Interlisp functions and variables discussed in the text as well as other topics of interest to the reader. Functions are indicated by a list of their arguments enclosed in parentheses after their name. The numbers in square brackets after a function or variable name indicate the chapter, section, or subsection in which that function or variable is described or defined. For example, COPY [6.4.1] is discussed in section 6.4.1.