XEROX

Interlisp-D Reference Manual
Volume I: Language

3101272
October, 1985

TABLE OF CONTENTS

[This page intentionally left blank]

BACKGROUND AND ACKNOWLEDGEMENTS

1 A Brief History of Interlisp

Interlisp began with an implementation of the Lisp programming language for the PDP-1 at Bolt, Beranek and Newman in 1966. It was followed in 1967 by 940 Lisp for the SDS-940 computer, which was the first Lisp system to use software paging techniques and a large virtual memory in conjunction with a list-processing system [Bobrow & Murphy, 1967]. DWIM, the Do-What-I-Mean error correction facility, was introduced into this system in 1968 by Warren Teitelman [Teitelman, 1969].

In 1970 BBN-Lisp, an upward compatible Lisp system for the PDP-10, was implemented under the Tenex operating system [Teitelman, et al., 1972]. With the hardware paging and 256K of virtual memory provided by Tenex, it was practical to provide more extensive and sophisticated user support facilities, and a library of such facilities began to evolve. In 1972, the name of the system was changed to Interlisp, and its development became a joint effort of the Xerox Palo Alto Research Center and Bolt, Beranek and Newman. The next few years saw a period of rapid growth and development of the language, the system and the user support facilities, including the record package, the file package, and Masterscope.

In 1974, an implementation of Interlisp was begun for the Xerox Alto, an experimental microprogrammed personal computer [Thacker et al., 1979]. AltoLisp [Deutsch, 1973] introduced the idea of providing a specialized, microcoded instruction set that modelled the basic operations of Lisp more closely than a general-purpose instruction set could -- and as such was the first true "Lisp machine". AltoLisp also served as a departure point for Interlisp-D, the implementation of Interlisp for the Xerox 1100 Series of personal computers, which was begun in 1979 [Sheil & Masinter, 1983].

In 1976, partially as a result of the AltoLisp effort, a specification for the Interlisp "virtual machine" was published [Moore, 1976]. This attempted to specify a small set of "primitive" operations which would support all of the higher level user facilities, which were nearly all written in Lisp. Although incomplete and written at a level which preserved too many of the details of the Tenex operating system, this document proved to be a watershed in the development of Interlisp, since it gave a clear definition of a (relatively) small kernel whose implementation would suffice to port Interlisp to a new environment. This was decisive in enabling many subsequent implementations.

Most recently, the implementation of Interlisp on personal workstations has extended Interlisp in major ways. Most striking has been the incorporation of interactive graphics and local area network facilities. Not only have these extensions expanded the range of applications for which Interlisp is being used, but the personal machine capabilities have had a major impact on the Interlisp programming system itself. Whereas the original Interlisp user interface assumed a very limited (teletype) channel to the user, the use of interactive graphics and the "mouse" pointing device has radically expanded the bandwidth of communication between the user and the machine. This has enabled completely new styles of interaction with the user (e.g., the use of multiple windows to provide several different interaction channels with the user) and these have provided both new programming tools and new ways of viewing and using the existing ones. In addition, the increased use of local area networks (such as the Ethernet) has expanded the horizon of the Interlisp user beyond the local machine to a whole community of machines, processes and services. Large portions of this manual are devoted to documenting the enhanced environment that has resulted from these developments.

2. Interlisp Implementations

Development of Interlisp for the PDP-10 was, until approximately 1978, funded by the Advanced Research Projects Administration of the Department of Defence (DARPA). Subsequent developments, which have emphasized the personal workstation facilities, have been sponsored by the Xerox Corporation, with contributions from members of the Interlisp user community.

Although there are a variety of implementations of Interlisp in use, this manual is a reference manual for the Interlisp-D implementation. Notes may occasionally be included on other implementations, but there is no guarantee that this information is complete for implementations other than Interlisp-D. For some implementations, there is a "Users Guide" which documents features which are completely unique to that machine; for example, how to turn on the system, logging on, and unique facilities which link Interlisp to the host environment or operating system.

3 Acknowledgements

The Interlisp system is the work of many people -- after nearly twenty years, too many even to list, much less detail their contributions. Nevertheless, some individuals cannot go unacknowledged:

Warren Teitelman, more than anyone else, made Interlisp "happen". Warren designed and implemented large parts of several generations of Interlisp, including the initial versions of most of the user facilities, coordinated the system development and assembled and edited the first four editions of the Interlisp reference manual.

Larry Masinter is a principal architect of the current Interlisp system, has contributed extensively to several implementations, and has designed and developed major extensions to both the Interlisp language and the programming environment.

Dan Bobrow was a principal designer of Interlisp's predecessors, has contributed to the implementation of several generations of Interlisp, and (in collaboration with others) made major advances in the underlying architecture, including the spaghetti stack, the transaction garbage collector, and the block compiler.

Ron Kaplan has decisively shaped many of the programming language extensions and user facilities of Interlisp, has played a key role in two implementations and has contributed extensively to the design and content of the Interlisp reference manual.

Peter Deutsch designed the AltoLisp implementation of Interlisp which developed several key design insights on which the current generation of personal machine implementations depends.

No matter where one ends this list, one is tempted to continue. Many others who contributed to particular implementations or revisions are acknowledged in the documentation for those systems. Following that tradition, this manual, which primarily documents the Interlisp-D implementation, acknowledges, in addition to those listed above, the work of:

Bill van Melle, who designed and implemented most of the local area network facilities, the process mechanism, and much of the run time support system.

Richard Burton, who designed and implemented a great deal of the interactive display facilities.

and the contributions of Alan Bell, Don Charnley, Mitch Lichtenberg, Steve Purcell, Eric Schoen, Beau Sheil, John Sybalsky, and the many others who have helped and contributed to the development of Interlisp-D.

Like Interlisp itself, the Interlisp Reference Manual is the work of many people, some of whom are acknowledged above. This edition was substantially rewritten, designed, edited and produced by Michael Sannella of Xerox Artificial Intelligence Systems. It is a major revision of the previous edition --- it has been completely reorganized, updated in most sections, and extended with a large amount of new material.

Interlisp is not designed by a formal committee. It grows and changes in response to the needs of those who use it. Contributions and discussion from the user community remain, as they always have been, warmly welcome.

4 References

[Bobrow & Murphy, 1967] Bobrow, D.G., and Murphy, D.L., "The Structure of a LISP System Using Two Level Storage" --- *Communications of the ACM*, Vol. 10, 3, (March, 1967).

[Deutsch, 1973] Deutsch, L.P., "A Lisp machine with very compact programs" --- *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford, (1973).

[Moore, 1976] Moore, J.S., "The Interlisp Virtual Machine Specification" --- Xerox PARC, CSL-76-5, (1976).

[Sheil & Masinter, 1983] Sheil, B., and Masinter, L.M. (eds.), "Papers on Interlisp-D" --- Xerox PARC, CIS-5 (Revised), (1983).

[Teitelman, 1969] Teitelman, W., "Toward a Programming Laboratory" --- *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, (1969).

[Teitelman, et al., 1972] Teitelman, W., Bobrow, D.G., Hartley, A.K. Murphy, D.L., *BBN-LISP TENEX Reference Manual* --- Bolt Beranek and Newman, (July 1971, first revision February 1972, second revision August 1972).

[Thacker, et al., 1979] Thacker, C., Lampson, B., and Sproull, R., "Alto: A personal computer" --- Xerox PARC,CSL-79-11, (August, 1979).

[This page intentionally left blank]

1. INTRODUCTION

Interlisp is a *programming system*. A programming system consists of a programming *language*, a large number of predefined programs (or *functions*, to use the Lisp terminology) that can be used either as direct user commands or as subroutines in user programs, and an *environment* that supports the programmer by providing a variety of specialized programming tools. The language and predefined functions of Interlisp are rich, but similar to those of other modern programming languages. The Interlisp programming environment, on the other hand, is very distinctive. Its most salient characteristic is an integrated set of programming tools which know enough about Interlisp programming so that they can act as semi-autonomous, intelligent "assistants" to the programmer. In addition, the environment provides a completely self-contained world for creating, debugging and maintaining Interlisp programs.

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

Whereas the rest of this manual describes the individual pieces of the Interlisp system, this chapter attempts to describe the whole system---language, environment, tools, and the otherwise unstated philosophies that tie it all together. It is intended to give a global view of Interlisp to readers approaching it for the first time.

1.1 Interlisp as a Programming Language

This manual does not contain an introduction to programming in Lisp. In this section, we simply highlight a few key points about Lisp on which much of the later material depends.

The Lisp family of languages shares a common structure in which large programs (or functions) are built up by composing the results of smaller ones. Although Interlisp, like most modern Lisps, allows programming in almost any style one can imagine, the natural style of Lisp is functional and recursive, in that each function computes its result by selecting from or building upon the values given to it and then passing that result back to its caller (rather than by producing "side-effects" on external data structures, for example). A great many applications can be written in Lisp in this purely functional style, which is encouraged by the simplicity with which Lisp functions can be composed together.

Lisp is also a list-manipulation language. The essential primitive data objects of any Lisp are "atoms" (symbols or identifiers) and "lists" (sequences of atoms or lists), rather than the "characters" or "numbers" of more conventional programming languages (although these are also present in all modern Lisps). Each Lisp dialect has a set of operations that act on atoms and lists, and these operations comprise the core of the language.

Invisible in the programs, but essential to the Lisp style of programming, is an automatic memory management system (an "allocator" and a "garbage collector"). Allocation of new storage occurs automatically whenever a new data object is created. Conversely, that storage is automatically reclaimed for reuse when no other object makes reference to it. Automatic allocation and deallocation of memory is essential for rapid, large scale program development because it frees the programmer from the task of maintaining the details of memory administration, which change constantly during rapid program evolution.

A key property of Lisp is that it can represent Lisp function definitions as pieces of Lisp list data. Each subfunction "call" (or *function application*) is written as a list in which the function is written first, followed by its arguments. Thus, **(PLUS 1 2)** is a list structure representation of the expression **1 + 2**. Each program can be written as a list of such function applications. This representation of program as data allows one to apply the same operations to programs that one uses to manipulate data, which makes it very straightforward to write Lisp programs which look at and change *other Lisp programs*. This, in turn, makes it easy to develop programming tools and translators, which was essential in enabling the development of the Interlisp environment.

One result of this ability to have one program examine another is that one can extend the Lisp programming language itself. If some desired programming idiom is not supported, it can be added simply by defining a function that translates the desired expression into simpler Lisp. Interlisp provides extensive facilities for users to make this type of language extension. Using this ability to extend itself, Interlisp has incorporated many of the constructs that have been developed in other modern programming languages (e.g. if-then-else, do loops, etc.).

1.2 Interlisp as an Interactive Environment

Interlisp programs should not be thought of as autonomous, external files of source code. All Interlisp programming takes place within the Interlisp environment, which is a completely self-sufficient environment for developing and using Interlisp programs. Not only does the environment contain the obvious programming facilities (e.g., program editors, compilers, debuggers, etc.), but it also contains a variety of tools which assist the user by "keeping track" of what happens, so the user doesn't have to. For example, the Interlisp file package notices when programs or data have been changed, so that the system will know what needs to be saved at the end of the session. The "residential" style, where one stays within the environment throughout the development, from initial program definition through final debugging, is essential for these tools to operate. Furthermore, this same environment is available to support the final production version, some parts providing run time support and other parts being ignored until the need arises for further debugging or development.

For terminal interaction with the user, Interlisp provides a top level "Read-Eval-Print" executive, which reads whatever the user types in, evaluates it, and prints the result. (This interaction is also recorded by the programmer's assistant, described below, so the user can ask to do an action again, or even to undo the effects of a previous action.) Although each interactive executive defines a few specialized commands, most of the interaction will consist of simple evaluations of ordinary Lisp expressions. Thus, instead of specialized terminal commands for operations like manipulating the user's files, actions like this are carried out simply by typing the same expressions that one would use to accomplish them inside a Lisp program. This creates a very rich, simple and uniform set of interactive commands, since any Lisp expression can be typed at a command executive and evaluated immediately.

In normal use, one writes a program (or rather, "defines a function") simply by typing in an expression that invokes the "function defining" function (**DEFINEQ**), giving it the name of the function being defined and its new definition. The newly defined function can be executed immediately, simply by using it in a Lisp expression. Although most Interlisp code is normally run compiled (for reasons of efficiency), the initial versions of most programs, and all of the user's terminal interactions, will be run interpreted. Eventually, as a function gets larger or is used in many places, it becomes more effective to compile it. Usually, by that stage, the function has been stored on a file and the whole file (which may contain many functions) is compiled at once. **DEFINEQ**, the compiler **(COMPILE)**, and the interpreter **(EVAL)**, are all themselves Lisp functions that use the ability to treat other Lisp expressions and programs as data.

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

List structure editor — Since Interlisp programs are represented as list structure, Interlisp provides an editor which allows one to change the list structure of a function's definition directly. See page 16.1

Pretty-printer — The pretty printer is a function that prints Lisp function definitions so that their syntactic structure is displayed by the indentation and fonts used. See page 26.40.

Break Package — When errors occur, the break package is called, allowing the user to examine and modify the context at the point of the error. Often, this enables execution to continue without starting over from the beginning. Within a break, the full power of Interlisp is available to the user. Thus, the broken function can be edited, data structures can be inspected and changed, other computations carried out, and so on. All of this occurs in the context of the suspended computation, which will remain available to be resumed. See page 14.1.

DWIM — The "Do What I Mean" package automatically fixes the user's misspellings and errors in typing. See page 20.1.

Programmer's Assistant — Interlisp keeps track of the user's actions during a session and allows each one to be replayed, undone, or altered. See page 13.1.

Masterscope — Masterscope is a program analysis and management tool which can analyze users' functions and build (and automatically maintain) a data base of the results. This allows the user to ask questions like "**WHO CALLS ARCTAN**" or "**WHO USES COEF1 FREELY**" or to request systematic changes like "**EDIT WHERE ANY** *(function)* **FETCHES ANY FIELD OF** *(the data structure)* **FOO**". See page 19.1.

Record/Datatype Package — Interlisp allows a programmer to define new data structures. This enables one to separate the issues of data access from the details of how the data is actually stored. See page 8.1.

File Package — Files in Interlisp are managed by the system, removing the problem of ensuring timely file updates from the user. The file package can be modified and extended to accomodate new types of data. See page 17.1.

Performance Analysis — These tools allow statistics on program operation to be collected and analyzed. See page 22.1.

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

The personal machine implementations of Interlisp, such as Interlisp-D, provide some additional facilities, and interactive graphic interfaces to some of the older Interlisp programming tools:

Multiple Processes — Multiple and independent processes simplify problems which require logically separate pieces of code to operate in parallel. See page 23.1.

Windows — The ability to have multiple, independent windows on the display allows many different processes or activities to be active on the screen at once. See page 28.2.

Inspector — The inspector is a display tool for examining complex data structures encountered during debugging. See page 26.1.

Interlisp-D has embedded within it an entire operating system written in Interlisp. For the most part, that is of no concern to the user (although it is nice to know that one *can* write programs of this complexity and performance within Interlisp!). However, some of the facilities provided by this low level code allow the use of Interlisp for applications that would previously have been forced into a relatively impoverished system programming environment. In particular, Interlisp-D provides complete facilities for experimenting with distributed machines and services on a local area network, plus access to all the services that such networks provide (e.g., mail, printing, filing, etc.).

1.3 Interlisp Philosophy

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

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

Another way in which users can effect Interlisp's behavior is by redefining or changing system functions. The "Advise" capability, for instance, permits the user to modify the operation of virtually any function in the system by wrapping user code "around" the selected function. (This same philosophy extends to the break package and tracing, so almost any function in the system can be broken or traced.) Experimentation is thus encouraged and actively facilitated, which allows the user to find useful pieces of the Interlisp system which can be configured to assist with application development. Since the entire system is implemented in Interlisp, there are extremely few places where the system's behavior depends on anything that the user cannot modify (such as a low level system implementation language).

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

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

1.4 How to Use this Manual

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

Users should not try to read straight through this manual, like a novel. In general, the chapters are organized with overview explanations and the most useful functions at the beginning of the chapter, and implementation details towards the end. If you are interested in becoming acquainted with Interlisp using this manual, the best way would be to skim through the whole book, reading the beginning of each chapter.

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

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

Case is significant: An important piece of information, often missed by newcomers to Interlisp, is that *upper and lower case is significant.* The variable **FOO** is not the same as the variable **foo** or the variable **Foo**. By convention, most Interlisp system functions and variables are all uppercase, but users are free to use upper and lower case for their own functions and variables as they wish.

One exception to the case-significance rule is provided by the Interlisp CLISP facility, which allows iterative statement operators and record operations to be typed in either all uppercase or all lowercase letters: **(for X from 1 to 5 ...)** is the same as **(FOR X FROM 1 TO 5 ...)**. The few situations where this is the case are explicitly mentioned in the manual. Generally, one should assume that case is significant.

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

(FOO *BAR BAZ* **—)** [Function]

This is a description for the function named **FOO**. **FOO** has two arguments, *BAR* and *BAZ*. Some system functions have extra optional arguments that are not documented and should not be used. These extra arguments are indicated by "—".

The descriptor [Function] indicates that this is a function, rather than a [Variable], [Macro], etc. For function definitions only, this can also indicate the "function type" (see page 10.2): [NLambda Function], [NoSpread Function], or [NLambda NoSpread Function], which describes whether the function takes a fixed or variable number of arguments, and whether the arguments are evaluated or not. [Function] indicates a lambda spread function (the most common function type).

1.5 References

[Burton, et al., 1980] Burton, R. R., L. M. Masinter, A. Bell, D. G. Bobrow, W. S. Haugeland, R.M. Kaplan and B.A. Sheil, "Interlisp-D: Overview and Status" --- in [Sheil & Masinter, 1983].

[Sandewall, 1978] Sandewall, Erik, "Programming in the Interactive Environmnet: The LISP Experience" --- *ACM Computing Surveys*, vol 10, no 1, pp 35-72, (March 1978).

[Sheil, 1983] Sheil, B.A., "Environments for Exploratory Programming" --- *Datamation*, (February, 1983) --- also in [Sheil & Masinter, 1983].

[Sheil & Masinter, 1983] Sheil, B.A. and L. M. Masinter, "Papers on Interlisp-D", Xerox PARC Technical Report CIS-5 (Revised), (January, 1983).

[Teitelman & Masinter, 1981] Teitelman, W. and L. M. Masinter, "The Interlisp Programming Environment" --- *Computer*, vol 14, no 4, pp 25-34, (April 1981) --- also in [Sheil & Masinter, 1983].

[This page intentionally left blank]

2. LITATOMS

A "litatom" (for "literal atom") is an object which conceptually consists of a print name, a value, a function definition, and a property list. In some Lisp dialects, litatoms are also known as "symbols."

A litatom is read as any string of non-delimiting characters that cannot be interpreted as a number. The syntatic characters that delimit litatoms are called separator or break characters (see page 25.33) and normally are space, end-of-line, line-feed, ((left paren),) (right paren), " (double quote), [(left bracket), and] (right bracket). However, any character may be included in a litatom by preceding it with the character %. Here are some examples of litatoms:

A wxyz 23SKIDDOO %] 3.1415+17

Long%Litatom%With%Embedded%Spaces

(LITATOM *X*) [Function]

Returns **T** if *X* is a litatom, **NIL** otherwise. Note that a number is not a litatom.

(LITATOM NIL) = T.

(ATOM *X*) [Function]

Returns **T** if *X* is an atom (i.e. a litatom or a number); **NIL** otherwise.

Warning: **(ATOM *X*)** is **NIL** if *X* is an array, string, etc. In many dialects of Lisp, the function **ATOM** is defined equivalent to the Interlisp function **NLISTP**.

(ATOM NIL) = T.

Litatoms are printed by **PRINT** and **PRIN2** as a sequence of characters with %'s inserted before all delimiting characters (so that the litatom will read back in properly). Litatoms are printed by **PRIN1** as a sequence of characters without these extra %'s. For example, the litatom consisting of the five characters **A**, **B**, **C**, **(**, and **D** will be printed as **ABC%(D** by **PRINT** and **ABC(D** by **PRIN1**.

Litatoms can also be constructed by **PACK**, **PACK***, **SUBATOM**, **MKATOM**, and **GENSYM** (which uses **MKATOM**).

Litatoms are unique. In other words, if two litatoms print the same, they will *always* be **EQ**. Note that this is *not* true for strings, large integers, floating point numbers, and lists; they all can print the same without being **EQ**. Thus if **PACK** or **MKATOM** is given a list of characters corresponding to a litatom that already exists, they return a pointer to that litatom, and do *not* make a new litatom. Similarly, if the read program is given as input a sequence of characters for which a litatom already exists, it returns a pointer to that litatom. Note: Interlisp is different from other Lisp dialects which allow "uninterned" litatoms.

Note: Litatoms are limited to 255 characters in Interlisp-D; 127 characters in Interlisp-10. Attempting to create a larger litatom either via **PACK** or by typing one in (or reading from a file) will cause an error, **ATOM TOO LONG**.

2.1 Using Litatoms as Variables

Litatoms are commonly used as variables. Each litatom has a "top level" variable binding, which can be an arbitrary Interlisp object. Litatoms may also be given special variable bindings within **PROG**s or function calls, which only exist for the duration of the function. When a litatom is evaluated, the "current" variable binding is returned. This is the most recent special variable binding, or the top level binding if the litatom has not been rebound. **SETQ** is used to change the current binding. For more information on variable bindings in Interlisp, see page 11.1.

Note: The compiler (page 18.1) treats variables somewhat differently than the interpreter, and the user has to be aware of these differences when writing functions that will be compiled. For example, variable references in compiled code are not checked for **NOBIND**, so compiled code will not generate unbound atom errors. In general, it is better to debug interpreted code, before compiling it for speed. The compiler offers some facilities to increase the efficiency of variable use in compiled functions. Global variables (page 18.4) can be defined so that the entire stack is not searched at each variable reference. Local variables (page 18.5) allow compiled functions to access variable bindings which are not on the stack, which reduces variable conflicts, and also makes variable lookup faster.

By convention, a litatom whose top level binding is to the litatom **NOBIND** is considered to have no top level binding. If a litatom has no local variable bindings, and its top level value is **NOBIND**, attempting to evaluate it will cause an unbound atom error.

The two litatoms **T** and **NIL** always evaluate to themselves. Attempting to change the binding of **T** or **NIL** with the functions below will generate the error **ATTEMPT TO SET T** or **ATTEMPT TO SET NIL**.

The following functions (except **BOUNDP**) will also generate the error **ARG NOT LITATOM**, if not given a litatom.

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

Returns **T** if *VAR* has a special variable binding (even if bound to **NOBIND**), or if *VAR* has a top level value other than **NOBIND**; otherwise **NIL**. In other words, if *X* is a litatom, **(EVAL** *X***)** will cause an **UNBOUND ATOM** error if and only if **(BOUNDP** *X***)** returns **NIL**.

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

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

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

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

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

Note: Since **SETQ** is an nlambda, *neither* argument is evaluated during the calling process. However, **SETQ** itself calls **EVAL** on its second argument. As a result, typing **(SETQ VAR FORM)** and **SETQ(VAR FORM)** to the Interlisp executive is equivalent: in both cases **VAR** is not evaluated, and **FORM** is.

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

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

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

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

(GETTOPVAL *VAR***)** [Function]

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

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

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

A major difference between various Interlisp implementations is the way that variable bindings are implemented. Interlisp-10 and Interlisp-Jerico use what is called "shallow" binding. Interlisp-D and Interlisp-VAX use what is called "deep" binding.

In a deep binding system, a variable is bound by saving on the stack the variable's new value. When a variable is accessed, its value is found by searching the stack for the most recent binding. If the variable is not found on the stack, the top level binding is retrieved from a "value cell" associated with the variable.

In a "shallow" binding system, a variable is bound by saving on the stack the variable name and the variable's old value and putting the new value in the variable's value cell. When a variable is accessed, its value is always found in its value cell.

GETTOPVAL and **SETTOPVAL** are less efficient in a shallow binding system, because they have to search the stack for rebindings; it is more economical to simply rebind variables. In a deep binding system, **GETTOPVAL** and **SETTOPVAL** are very efficient since they do not have to search the stack, but can simply access the value cell directly.

GETATOMVAL and **SETATOMVAL** can be used to access a variable's value cell, in either a shallow or deep binding system.

(GETATOMVAL *VAR***)** [Function]

Returns the value in the value cell of *VAR*. In a shallow binding system, this is the same as **(EVAL** *ATM***)**, or simply *VAR*. In a deep binding system, this is the same as **(GETTOPVAL** *VAR***)**.

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

Sets the value cell of *VAR* to *VALUE*. In a shallow binding system, this is the same as **SET**; in a deep binding system, this is the same as **SETTOPVAL**.

2.2 Function Definition Cells

Each litatom has a function definition cell, which is accessed when a litatom is used as a function. The mechanism for accessing and setting the function definition cell of a litatom is described on page 10.9.

2.3 Property Lists

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

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

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

The functions below are used to manipulate the propert lists of litatoms. Except when indicated, they generate the error **ARG NOT LITATOM**, if given an object that is not a litatom.

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

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

Note: **GETPROP** used to be called **GETP**.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Property lists are conventionally implemented as lists of the form

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

although the user can store anything as the property list of a litatom. However, the functions which manipulate property lists observe this convention by searching down the property lists two **CDR**s at a time. Most of these functions also generate an error, **ARG NOT LITATOM**, if given an argument which is not a litatom, so they cannot be used directly on lists. (**LISTPUT**, **LISTPUT1**, **LISTGET**, and **LISTGET1** are functions similar to **PUTPROP** and **GETPROP** that work directly on lists. See page 3.16.) The property lists of litatoms can be directly accessed with the following functions:

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

Returns the property list of *ATM*.

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

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

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

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

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

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

2.4 Print Names

Each litatom has a print name, a string of characters that uniquely identifies that litatom. The term "print name" has been extended, however, to refer to the characters that are output when any object is printed. In Interlisp, all objects have print names, although only litatoms and strings have their print name explicitly stored. This section describes a set of functions which can be used to access and manipulate the print names of any object, though they are primarily used with the print names of litatoms.

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

Sometimes we will have occasion to refer to a "**PRIN2**-name." The **PRIN2**-name of an object is those characters output when the object is printed using **PRIN2**. Thus the **PRIN2**-name of the litatom **ABC%(D** is the *six* characters **ABC%(D**. Note that the **PRIN2**-name depends on what readtable is being used (see page 25.33), since this determines where %'s will be inserted. Many of the functions below allow either print names or **PRIN2**-names to be used, as specified by *FLG* and *RDTBL* arguments. If *FLG* is **NIL**, print names are used. Otherwise, **PRIN2**-names are used, computed with respect to the readtable *RDTBL* (or the current readtable, if *RDTBL* = **NIL**).

Note: The print name of an integer depends on the setting of **RADIX** (page 25.13). The functions described in this section (**UNPACK**, **NCHARS**, etc.) define the print name of an integer as though the radix was 10, so that **(PACK (UNPACK 'X9))** will always be **X9** (and not **X11**, if **RADIX** is set to 8). However, integers will still be *printed* by **PRIN1** using the current radix. The user can force these functions to use print names in the current radix by changing the setting of the variable **PRXFLG** (page 25.14).

(MKATOM *X*) [Function]

Creates and returns an atom whose print name is the same as that of the string *X* or, if *X* isn't a string, the same as that of **(MKSTRING *X*)**. Examples:

(MKATOM '(A B C)) => %(A%.B% C%)

(MKATOM "1.5") => 1.5

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

(SUBATOM *X N M*) [Function]

Equivalent to **(MKATOM (SUBSTRING *X N M*))**, but does not make a string pointer (see page 4.3). Returns an atom made from the *N*th through *M*th characters of the print name of *X*. If *N* or *M* are negative, they specify positions counting backwards from the end of the print name. Examples:

(SUBATOM "FOO1.5BAR" 4 6) => 1.5

(SUBATOM '(A B C) 2 -2) => A% B% C

(PACK *X*) [Function]

If *X* is a list of atoms, **PACK** returns a single atom whose print name is the concatenation of the print names of the atoms in *X*.

If the concatenated print name is the same as that of a number, **PACK** will return that number. For example,

(PACK '(A BC DEF G)) => ABCDEFG

(PACK '(1 3.4)) => 13.4

(PACK '(1 E -2)) => .01

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

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

If *X* is not a list or **NIL**, **PACK** generates an error, **ILLEGAL ARG**.

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

Nospread version of **PACK** that takes an arbitrary number of arguments, instead of a list. Examples:,

(PACK* 'A 'BC 'DEF 'G) => ABCDEFG

(PACK* 1 3.4) => 13.4

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

Returns the print name of *X* as a list of single-characters atoms, e.g.,

(UNPACK 'ABC5D) => (A B C 5 D)

(UNPACK "ABC(D") => (A B C %(D)

If *FLG* = **T**, the **PRIN2**-name of *X* is used (computed with respect to *RDTBL*), e.g.,

(UNPACK "ABC(D" T) => (%" A B C %(D %")

(UNPACK 'ABC%(D" T) => (A B C %% %(D)

Note: **(UNPACK** *X***)** performs *N* **CONS**es, where *N* is the number of characters in the print name of *X*.

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

A destructive version of **UNPACK** that does not perform any **CONS**es but instead reuses the list *SCRATCHLIST*. If the print name is too long to fit in *SCRATCHLIST*, **DUNPACK** will extend it. If *SCRATCHLIST* is not a list, **DUNPACK** returns **(UNPACK** *X FLG RDTBL***)**.

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

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

(NCHARS 'ABC) => 3

(NCHARS "ABC" T) => 5

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

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

Returns the *N*th character of the print name of *X* as an atom. *N* can be negative, in which case it counts from the end of the print name, e.g., -1 refers to the last character, -2 next to last, etc. If *N* is greater than the number of characters in the print name, or less than minus that number, or 0, **NTHCHAR** returns **NIL**. Examples:

(NTHCHAR 'ABC 2) => B

(NTHCHAR 15.6 2) => 5

(NTHCHAR 'ABC%(D -3 T) => %%

(NTHCHAR "ABC" 2) => B

(NTHCHAR "ABC" 2 T) => A

Note: **NTHCHAR** and **NCHARS** work much faster on objects that actually have an internal representation of their print name, i.e., litatoms and strings, than they do on numbers and lists, as they do not have to simulate printing.

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

Returns a lower case version of *X*. If *FLG* is **T**, the first letter is capitalized. If *X* is a string, the value of **L-CASE** is also a string. If *X* is a list, **L-CASE** returns a new list in which **L-CASE** is computed for each corresponding element and non-**NIL** tail of the original list. Examples:

(L-CASE 'FOO) => foo

(L-CASE 'FOO T) => Foo

(L-CASE "FILE NOT FOUND" T) => "File not found"

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

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

Similar to **L-CASE**, except returns the upper case version of *X*.

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

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

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

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

GENNUM [Variable]

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

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

(MAPATOMS *FN*) [Function]

Applies *FN* (a function or lambda expression) to every litatom in the system. Returns **NIL**

For example,

```
(MAPATOMS (FUNCTION (LAMBDA(X)
          (if (GETD X) then (PRINT X)]
```

will print every litatom with a function definition.

Note: In some implementations of Interlisp, unused litatoms may be garbage collected, which can effect the action of **MAPATOMS**.

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

APROPOS scans all litatoms in the system for those which have *STRING* as a substring and prints them on the terminal along with a line for each relevant item defined for each selected atom. Relevant items are (1) function definitions, for which only the arglist is printed, (2) dynamic variable values, and (3) non-null property lists. **PRINTLEVEL** (page 25.11) is set to **(3 . 5)** when **APROPOS** is printing.

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

If *QUIETFLG* is non-**NIL**, then no printing at all is done, but instead a list of the selected atoms is returned.

If *OUTPUT* is non-**NIL**, the printing will be directed to *OUTPUT* (which should be a stream open for output) instead of to the terminal stream.

2.5 Characters and Character Codes

Characters may be represented in two ways: as single-character atoms, or as integer character codes. In many situations, it is more efficient to use character codes, so Interlisp provides parallel functions for both representations.

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

Le système d'information Xerox 11xx est remarquablement polyglotte.

Das Xerox 11xx Kommunikationssystem bietet merkwürdige multilinguale Nutzmöglichkeiten.

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

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

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

In almost all cases, a program does not have to distinguish between NS characters or 8-bit characters. The exception to this rule is the handling of input/output operations (see page 25.22).

The function **CHARCODE** (page 2.13) provides a simple way to create individual NS characters codes. The VirtualKeyboards library package provides a set of virtual keyboards that allow keyboard or mouse entry of NS characters.

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

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

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

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

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

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

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

Similar to **DUNPACK**.

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

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

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

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

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

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

N is a character code. Returns the atom having the corresponding single character as its print name.

(CHARACTER 70) => F

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

Fast version of **CHARACTER** that compiles open.

The following function makes it possible to gain the efficiency that comes from dealing with character codes without losing the symbolic advantages of character atoms:

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

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

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

CR, SPACE, etc.

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

(CHARCODE SPACE) => 32

(CHARCODE CR) => 13

CHARSET,CHARNUM
CHARSET-CHARNUM

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

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

CHARNUM is either an *octal* number, a single-character litatom, or a litatom from the association list **CHARACTERNAMES**. Note that if *CHARNUM* is a single-digit number, it is interpreted as the character "2", rather than as the octal number 2. Examples:

(CHARCODE 12,6) => 2566

(CHARCODE 12,SPACE) => 2592

(CHARCODE GREEK,A) => 9793

↑ *CHARSPEC* (control chars)

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

(CHARCODE ↑A) => 1

(CHARCODE ↑GREEK,A) => 9729

#CHARSPEC (meta chars)

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

(CHARCODE #A) => 193

(CHARCODE #↑GREEK,A) => 9857

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

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

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

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

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

Similar to **SELECTQ** (page 9.6), except that the selection keys are determined by applying **CHARCODE** (instead of **QUOTE**) to the key-expressions. If the value of *E* is a character code or **NIL** and it is **EQ** or **MEMB** to the result of applying **CHARCODE** to the first element of a clause, the remaining forms of that clause are evaluated. Otherwise, the default is evaluated.

Thus

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

is exactly equivalent to

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

Furthermore, **SELCHARQ** has a macro definition such that it always compiles as an equivalent **SELECTQ**.

[This page intentionally left blank]

[This page intentionally left blank]

3. LISTS

One of the most useful datatypes in Interlisp is the list cell, a data structure which contains pointers to two other objects, known as the **CAR** and the **CDR** of the list cell (after the accessing functions). Very complicated structures can be built out of list cells, including lattices and trees, but list cells are most frequently used for representing simple linear lists of objects.

The following functions are used to manipulate list cells:

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

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

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

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

(LISTP NIL) = NIL.

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

(NOT (LISTP X)). Returns **T** if *X* is not a list cell, **NIL** otherwise.

(NLISTP NIL) = T.

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

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

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

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

CAR/CDRERR [Variable]

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

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

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

If **CAR/CDRERR** = **CDR**, then **CAR** of a non-list returns **"{car of non-list}"** as above, but **CDR** of a non-list causes an error. This setting is based on the observation that nearly all infinite loops involving non-lists occur from taking **CDR**s, but a fair amount of careless code takes **CAR** of something it has not tested to be a list

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

(CAADR X) = = > (CAR (CAR X))

(CADR X) = = > (CAR (CDR X))

(CDDDDR X) = = > (CDR (CDR (CDR (CDR X))))

All 30 combinations of nested **CAR**s and **CDR**s up to 4 deep are included in the system.

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

Replaces the **CDR** of the list cell *X* with *Y*. This physically changes the internal structure of *X*, as opposed to **CONS**, which creates a new list cell. It is possible to construct a circular list by using **RPLACD** to place a pointer to the beginning of a list in a spot at the end of the list.

The value of **RPLACD** is *X*. An attempt to **RPLACD NIL** will cause an error, **ATTEMPT TO RPLAC NIL** (except for **(RPLACD NIL NIL)**). An attempt to **RPLACD** any other non-list will cause an error, **ARG NOT LIST**.

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

Similar to **RPLACD**, but replaces the **CAR** of *X* with *Y*. The value of **RPLACA** is *X*. An attempt to **RPLACA NIL** will cause an error, **ATTEMPT TO RPLAC NIL**, (except for **(RPLACA NIL NIL)**). An attempt to **RPLACA** any other non-list will cause an error, **ARG NOT LIST**.

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

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

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

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

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

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

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

(FRPLNODE2 *X Y***)** [Function]

Faster versions of **RPLACD**, etc.

Usually, single list cells are not manipulated in isolation, but in structures known as "lists". By convention, a list is represented by a list cell whose **CAR** is the first element of the list, and whose **CDR** is the rest of the list (usually another list cell or the "empty list," **NIL**). List elements may be any Interlisp objects, including other lists.

The input syntax for a list is a sequence of Interlisp data objects (litatoms, numbers, other lists, etc.) enclosed in parentheses or brackets. Note that **()** is read as the litatom **NIL**. A right bracket can be used to match all left parenthesis back to the last left bracket, or terminate the lists, e.g. **(A (B (C]**.

If there are two or more elements in a list, the final element can be preceded by a period delimited on both sides, indicating that **CDR** of the final list cell in the list is to be the element immediately following the period, e.g. **(A . B)** or **(A B C . D)**, otherwise **CDR** of the last list cell in a list will be **NIL**. Note that a list does not have to end in **NIL**. It is simply a structure composed of one or more list cells. The input sequence **(A B C . NIL)** is equivalent to **(A B C)**, and **(A B . (C D))** is equivalent to **(A B C D)**. Note however that **(A B . C D)** will create a list containing the five litatoms **A**, **B**, **%.**, **C**, and **D**.

Lists are printed by printing a left parenthesis, and then printing the first element of the list, then printing a space, then printing the second element, etc. until the final list cell is reached. The individual elements of a list are printed by **PRIN1** if the list is being printed by **PRIN1**, and by **PRIN2** if the list is being printed by **PRINT** or **PRIN2**. Lists are considered to terminate when **CDR** of some node is not a list. If **CDR** of this terminal node is **NIL** (the usual case), **CAR** of the terminal node is printed followed by a right parenthesis. If **CDR** of the terminal node is *not* **NIL**, **CAR** of the terminal node is printed, followed by a space, a period, another space, **CDR** of the terminal node, and then the right parenthesis. Note that a list input as **(A B C . NIL)** will print as **(A B C)**, and a list input as **(A B . (C D))** will print as **(A B C D)**. Note also that **PRINTLEVEL** affects the printing of lists (page 25.11), and that carriage returns may be inserted where dictated by **LINELENGTH** (page 25.11).

Note: One must be careful when testing the equality of list structures. **EQ** will be true only when the two lists are the *exact* same list. For example,

```
←(SETQ A '(1 2))
(1 2)
←(SETQ B A)
(1 2)
←(EQ A B)
T
←(SETQ C '(1 2))
(1 2)
←(EQ A C)
NIL
←(EQUAL A C)
T
```

In the example above, the values of **A** and **B** are the exact same list, so they are **EQ**. However, the value of **C** is a totally different list, although it happens to have the same elements. **EQUAL** should be used to compare the elements of two lists. In general, one should notice whether list manipulation functions use **EQ** or **EQUAL** for comparing lists. This is a frequent source of errors.

Interlisp provides an extensive set of list manipulation functions, described in the following sections.

3.1 Creating Lists

(MKLIST *X*) [Function]

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

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

Returns a list of its arguments, e.g.

(LIST 'A 'B '(C D)) => (A B (C D))

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

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

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

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

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

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

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

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

The following examples illustrate the treatment of non-lists:

(APPEND '(A B C) 'D) => (A B C . D)

(APPEND 'A '(B C D)) => (B C D)

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

(APPEND '(A B C . D)) => (A B C . D)

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

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

Note that **NCONC** cannot change **NIL** to a list:

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

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

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

(NCONC *LST* **(LIST** *X***))**

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

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

3.2 Building Lists From Left to Right

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

TCONC is similar to **NCONC1**; it is useful for building a list by adding elements one at a time at the end. Unlike **NCONC1**, **TCONC** does not have to search to the end of the list each time it is called. Instead, it keeps a pointer to the end of the list being assembled, and updates this pointer after each call. This can be considerably faster for long lists. The cost is an extra list cell, *PTR*. **(CAR** *PTR***)** is the list being assembled, **(CDR** *PTR***)** is **(LAST (CAR** *PTR***))**. **TCONC** returns *PTR*, with its **CAR** and **CDR** appropriately modified.

PTR can be initialized in two ways. If *PTR* is **NIL**, **TCONC** will create and return a *PTR*. In this case, the program must set some variable to the value of the first call to **TCONC**. After that, it is unnecessary to reset the variable, since **TCONC** physically changes its value. Example:

```
←(SETQ FOO (TCONC NIL 1))
((1) 1)
←(for I from 2 to 5 do (TCONC FOO I))
NIL
←FOO
((1 2 3 4 5) 5)
```

If *PTR* is initially **(NIL)**, the value of **TCONC** is the same as for *PTR* = **NIL**. but **TCONC** changes *PTR*. This method allows the program to initialize the **TCONC** variable before adding any elements to the list. Example:

```
←(SETQ FOO (CONS))
(NIL)
←(for I from 1 to 5 do (TCONC FOO I))
NIL
←FOO
((1 2 3 4 5) 5)
```

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

Where **TCONC** is used to add *elements* at the end of a list, **LCONC** is used for building a list by adding *lists* at the end, i.e., it is similar to **NCONC** instead of **NCONC1**. Example:

```
←(SETQ FOO (CONS))
(NIL)
←(LCONC FOO '(1 2))
((1 2) 2)
←(LCONC FOO '(3 4 5))
((1 2 3 4 5) 5)
←(LCONC FOO NIL)
((1 2 3 4 5) 5)
```

LCONC uses the same pointer conventions as **TCONC** for eliminating searching to the end of the list, so that the same pointer can be given to **TCONC** and **LCONC** interchangeably. Therefore, continuing from above,

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

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

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

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

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

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

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

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

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

(SCRATCHLIST *LST* X_1 X_2 ... X_N**)** [NLambda NoSpread Function]

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

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

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

3.3 Copying Lists

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

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

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

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

Like **COPY** except copies down to atoms. Arrays, hash-arrays, strings, user data types, etc., are all copied. Analagous to **EQUALALL** (page 9.3). Note that this will not work if given a data structure with circular pointers; in this case, use **HCOPYALL**.

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

Similar to **COPYALL**, except that it will work even if the data structure contains circular pointers.

3.4 Extracting Tails of Lists

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

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

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

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

Returns the tail of *X* beginning with the *N*th element. Returns **NIL** if *X* has fewer than *N* elements. Examples:

(NTH '(A B C D) 1) => (A B C D)

(NTH '(A B C D) 3) => (C D)

(NTH '(A B C D) 9) => NIL

(NTH '(A . B) 2) => B

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

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

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

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

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

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

(LAST '(A B C)) => (C)

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

(LAST 'A) => NIL

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

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

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

NLEFT returns the tail of *L* that contains *N* more elements than *TAIL*. If *L* does not contain *N* more elements than *TAIL*, **NLEFT** returns **NIL**. If *TAIL* is **NIL** or not a tail of *L*, **NLEFT** returns the last *N* list cells in *L*. **NLEFT** can be used to work backwards through a list. Example:

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

←(NLEFT FOO 3 (CDDR FOO))
NIL

(LASTN *L N*) [Function]

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

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

(LASTN '(A B) 2) => (NIL A B)

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

3.5 Counting List Cells

(LENGTH *X*) [Function]

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

(LENGTH '(A B C)) => 3

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

(LENGTH 'A) => 0

(FLENGTH *X*) [Function]

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

(EQLENGTH *X N*) [Function]

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

(COUNT *X*) [Function]

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

(COUNT '(A)) => 1

(COUNT '(A . B)) => 1

(COUNT '(A (B) C)) => 4

In this last example, the value is 4 because the list **(A *X* C)** uses 3 list cells for any object *X*, and **(B)** uses another list cell.

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

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

(COUNTDOWN '(A) 100) => 99

(COUNTDOWN '(A . B) 100) => 99

(COUNTDOWN '(A (B) C) 100) => 96

(COUNTDOWN (DOCOLLECT 1 NIL) 100) => 0

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

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

(EQUALN '(((A)) B) '(((Z)) B) 2) => ?

(EQUALN '(((A)) B) '(((Z)) B) 3) => NIL

(EQUALN '(((A)) B) '(((A)) B) 3) => T

3.6 Logical Operations

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

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

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

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

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

Note that **(INTERSECTION X X)** gives a list of all members of **X** without any duplications.

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

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

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

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

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

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

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

Example:

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

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

3.7 Searching Lists

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

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

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

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

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

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

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

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

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

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

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

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

3.8 Substitution Functions

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

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

(SUBST 'A 'B '(C B (X . B))) => (C A (X . A))

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

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

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

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

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

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

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

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

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

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

ALST is a list of pairs:

$((OLD_1 \,.\, NEW_1)\ (OLD_2 \,.\, NEW_2)\ \ldots\ (OLD_N \,.\, NEW_N))$

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

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

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

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

Similar to **SUBLIS**, except it changes the list structure *EXPR* itself instead of copying it.

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

Similar to **SUBLIS**, except that elements of *NEW* are substituted for corresponding atoms of *OLD* in *EXPR*, e.g.,

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

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

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

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

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

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

3.9 Association Lists and Property Lists

A commonly-used data structure is one that associates an arbitrary set of property names (*NAME1*, *NAME2*, etc.), with a set of property values (*VALUE1*, *VALUE2*, etc.). Two list structures commonly used to store such associations are called "property lists" and "association lists." A list in "association list" format is a list where each element is a dotted pair whose **CAR** is a property name, and whose **CDR** is the value:

((*NAME1* . *VALUE1*) (*NAME2* . *VALUE2*) ...)

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

(*NAME1 VALUE1 NAME2 VALUE2* ...)

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

Note: Property lists are contained within many Interlisp-D system datatypes. There are special functions that can be used to set and retrieve values from the property lists of litatoms (see page 2.5), from properties of windows (see page 28.13), etc.

Note: Another data structure that offers some of the advantages of association lists and property lists is the hash array data type (see page 6.1).

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

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

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

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

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

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

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

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

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

Note that the argument order for **ASSOC**, **PUTASSOC**, etc. is different from that of **LISTGET**, **LISTPUT**, etc.

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

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

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

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

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

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

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

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

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

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

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

Note: **LISTGET1** used to be called **GET**.

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

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

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

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

```
←(SETQ FOO NIL)
```

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

3.10 Sorting Lists

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

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

The value of **SORT** is the sorted list. The sort is destructive and uses no extra storage. The value returned is **EQ** to *DATA* but elements have been switched around. Interrupting with control D, E, or B may cause loss of data, but control H may be used at any time, and **SORT** will break at a clean state from which ↑ or control characters are safe. The algorithm used by **SORT** is such that the maximum number of compares is $N*\log_2 N$, where N is **(LENGTH** *DATA***)**.

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

For example, if **(FOO . FIE)** appears before **(FOO . FUM)** in **X**, **(SORT X T)** may or may not reverse the order of these two elements. Of course, the user can always specify a more precise *COMPAREFN*.

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

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

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

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

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

If *CASEARRAY* is non-**NIL**, it is a casearray (page 25.21 that the characters of *A* and *B* are translated through before being compared. Note that numbers are not passed through *CASEARRAY*.

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

Note: **ALPHORDER** does no **UNPACK**s, **CHCON**s, **CONS**es or **NTHCHAR**s. It is several times faster for alphabetizing than anything that can be written using these other functions.

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

Defined as **(ALPHORDER** *A B* **UPPERCASEARRAY)**. **UPPERCASEARRAY** (page 25.22) is a casearray that maps every lowercase character into the corresponding uppercase character.

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

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

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

Returns *LST*. **MERGEINSERT** is undoable.

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

MERGEINSERT is used by **ADDTOFILE** (page 17.48) to insert the name of a new function into a list of functions. The algorithm is essentially to look for the item with the longest common leading sequence of characters with respect to *NEW*, and then merge *NEW* in starting at that point.

3.11 Other List Functions

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

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

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

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

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

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

Note that **DREMOVE** cannot *change* a list to **NIL**:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(COMPARELISTS '(A B C D) '(A D))
(A B C --)
(A D)

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

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

(NEGATE *X*) [Function]

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

(NEGATE '(MEMBER X Y)) => (NOT (MEMBER X Y))

(NEGATE '(EQ X Y)) => (NEQ X Y)

(NEGATE '(AND X (NLISTP X))) => (OR (NULL X) (LISTP X))

(NEGATE NIL) => T

[This page intentionally left blank]

4. STRINGS

A string is an object which represents a sequence of characters. Interlisp provides functions for creating strings, concatenating strings, and creating sub-strings of a string.

The input syntax for a string is a double quote ("), followed by a sequence of any characters except double quote and %, terminated by a double quote. The % and double quote characters may be included in a string by preceding them with the character %.

Strings are printed by **PRINT** and **PRIN2** with initial and final double quotes, and %s inserted where necessary for it to read back in properly. Strings are printed by **PRIN1** without the delimiting double quotes and extra %s.

A "null string" containing no characters is input as "". The null string is printed by **PRINT** and **PRIN2** as "". **(PRIN1 "")** doesn't print anything.

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

(STRINGP *X*) [Function]

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

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

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

(STREQUAL "ABC" "ABC") => T

(EQ "ABC" "ABC") => NIL

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

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

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

(STRING-EQUAL "FOO" "Foo") => T

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

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

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

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

If *FATFLG* is non-**NIL**, the string is allocated using full 16-bit NS characters (see page 2.12) instead of 8-bit characters. This can speed up some string operations if NS characters are later inserted into the string. This has no other effect on the operation of the string functions.

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

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

(MKSTRING "ABC") => "ABC"

(MKSTRING '(A B C)) => "(A B C)"

(MKSTRING NIL) => "NIL"

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

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

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

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

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

(NCHARS 'ABC) => 3

(NCHARS "ABC" T) => 5

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

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

Returns the substring of *X* consisting of the *N*th through *M*th characters of *X*. If *M* is **NIL**, the substring contains the *N*th character thru the end of *X*. *N* and *M* can be negative numbers, which are interpreted as counts back from the end of the string, as with **NTHCHAR** (page 2.10). **SUBSTRING** returns **NIL** if the substring is not well defined, e.g., *N* or *M* specify character positions outside of *X*, or *N* corresponds to a character in *X* to the right of the character indicated by *M*). Examples:

(SUBSTRING "ABCDEFG" 4 6) => "DEF"

(SUBSTRING "ABCDEFG" 3 3) => "C"

(SUBSTRING "ABCDEFG" 3 NIL) => "CDEFG"

(SUBSTRING "ABCDEFG" 4 -2) => "DEF"

(SUBSTRING "ABCDEFG" 6 4) => NIL

(SUBSTRING "ABCDEFG" 4 9) => NIL

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

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

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

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

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

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

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

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

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

```
←(SETQ FOO "ABCDEFG")
"ABCDEFG"
```

←(GLC FOO)
G
←(GLC FOO)
F
←FOO
"ABCDE"

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

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

(CONCAT "ABC" 'DEF "GHI") => "ABCDEFGHI"

(CONCAT '(A B C) "ABC") => "(A B C)ABC"

(CONCAT) returns the null string, "".

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

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

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

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

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

(RPLSTRING "ABCDEF" -3 "END") => "ABCEND"

(RPLSTRING "ABCDEFGHIJK" 4 '(A B C)) => "ABC(A B C)K"

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

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

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

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

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

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

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

(STRPOS "ABC" "XYZABCDEF") => 4

(STRPOS "ABC" "XYZABCDEF" 5) => NIL

(STRPOS "ABC" "XYZABCDEFABC" 5) => 10

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

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

(STRPOS "DEF&" "XYZABCDEF" NIL '&) => NIL

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

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

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

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

(STRPOS "ABC" "XYZABCDEFABC" NIL NIL NIL T) => 7

(STRPOS "A" "A" NIL NIL NIL T) => 2

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

If *CASEARRAY* is non-**NIL**, this should be a casearray like that given to **FILEPOS** (page 25.20). The casearray is used to map the string characters before comparing them to the search string.

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

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

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

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

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

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

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

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

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

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

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

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

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

TABLE OF CONTENTS

[This page intentionally left blank]

5. ARRAYS

An array in Interlisp is an object representing a one-dimensional vector of objects. Arrays are generally created by the function **ARRAY**.

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

Creates and returns a new array capable of containing *SIZE* objects of type *TYPE*. If *TYPE* is **NIL**, the array can contain any arbitrary Lisp datum. In general, *TYPE* may be any of the various field specifications which are legal in **DATATYPE** declarations (see page 8.9): **POINTER**, **FIXP**, **FLOATP**, **(BITS** *N***)**, etc. The implementation will, if necessary, choose an "enclosing" type if the given one is not supported; for example, an array of **(BITS 3)** may be represented by an array of **(BITS 8)**.

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

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

Note: Arrays of type **FLOATP** are stored unboxed. This increases the space and time efficiency of **FLOATP** arrays. Users who want to use boxed floating point numbers should use an array of type **POINTER** instead of **FLOATP**.

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

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

Note: In some implementations of Interlisp (but not Interlisp-D), **ARRAYP** may also return *X* if it is of type **CCODEP** or **HARRAYP**.

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

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

Generates the error **ARG NOT ARRAY** if *ARRAY* is not an array. Generates the error **ILLEGAL ARG** if *N* is out of bounds.

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

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

Generates the error **ARG NOT ARRAY** if *ARRAY* is not an array. Generates the error **ILLEGAL ARG** if *N* is out of bounds. Can generate the error **NON-NUMERIC ARG** if *ARRAY* is an array whose **ARRAYTYP** is **FIXP** or **FLOATP** and *VAL* is non-numeric.

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

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

Note: If **ARRAY** coerced the array type as described above, **ARRAYTYP** will return the *new* type. For example, **(ARRAYTYP (ARRAY 10 '(BITS 3)))** will return **BYTE** in Interlisp-D, and **FIXP** in Interlisp-10.

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

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

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

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

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

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

[This page intentionally left blank]

Hash arrays provide a mechanism for associating arbitrary lisp objects ("hash keys") with other objects ("hash values"), such that the hash value associated with a particular hash key can be quickly obtained. A set of associations could be represented as a list or array of pairs, but these schemes are very inefficient when the number of associations is large. There are functions for creating hash arrays, putting a hash key/value pair in a hash array, and quickly retrieving the hash value associated with a given hash key.

By default, the hash array functions use **EQ** for comparing hash keys. This means that if non-atoms are used as hash keys, the exact same object (not a copy) must be used to retrieve the hash value. However, the user can override this default for any hash array by specifying the functions used to compare hash keys and to "hash" a hash key to a number. This can be used, for example, to create hash arrays where **EQUAL** but non-**EQ** strings will hash to the same value. Specifying alternative hashing algorithms is described below (page 6.4).

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

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

(HASHARRAY *MINKEYS OVERFLOW HASHBITSFN EQUIVFN***)** [Function]

Creates a hash array containing at least *MINKEYS* hash keys, with overflow method *OVERFLOW*. See discussion of overflow behavior below (page 6.3).

If *HASHBITSFN* and *EQUIVFN* are non-**NIL**, they specify the hashing function and comparison function used to interpret hash keys. This is described in the section on user-specified hashing functions below (page 6.4). If *HASHBITSFN* and

EQUIVFN are **NIL**, the default is to hash **EQ** hash keys to the same value.

(HARRAY *MINKEYS***)** [Function]

Provided for backward compatibility, this is equivalent to **(HASHARRAY MINKEYS 'ERROR)**.

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

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

Note that **HARRAYP** returns **NIL** if *X* is a list whose **CAR** is an **HARRAYP**, even though this is accepted by the hash array functions.

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

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

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

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

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

Equivalent to **(HARRAYPROP HARRAY 'SIZE)**; returns the number of slots in *HARRAY*.

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

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

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

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

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

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

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

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

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

MAPHFN is a function of two arguments. For each hash key in *HARRAY*, *MAPHFN* will be applied to (1) the hash value, and (2) the hash key. For example,

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

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

(DMPHASH $HARRAY_1$ $HARRAY_2$ ... $HARRAY_N$**)** [NLambda NoSpread Function]

Prints on the primary output file **LOAD**able forms which will restore the hash-arrays contained as the values of the atoms $HARRAY_1$, $HARRAY_2$, ... $HARRAY_N$. Example: **(DMPHASH SYSHASHARRAY)** will dump the system hash-array.

Note: all **EQ** identities except atoms and small integers are lost by dumping and loading because **READ** will create new structure for each item. Thus if two lists contain an **EQ** substructure, when they are dumped and loaded back in, the corresponding substructures while **EQUAL** are no longer **EQ**. The **HORRIBLEVARS** file package command (page 17.36) provides a way of dumping hash tables such that these identities are preserved.

6.1 Hash Overflow

When a hash array becomes full, attempting to add another hash key will cause the function **HASHOVERFLOW** to be called. This will either automatically enlarge the hash array, or cause the error **HASH TABLE FULL**. How hash overflow is handled is determined by the value of the **OVERFLOW** property of the hash array (which can be accessed by **HARRAYPROP**). The possibilities for the overflow method are:

the litatom **ERROR** — The error **HASH ARRAY FULL** is generated when the hash array overflows. This is the default overflow behavior for hash arrays returned by **HARRAY**.

NIL — The array is automatically enlarged by 1.5. This is the default overflow behavior for hash arrays returned by **HASHARRAY**.

a positive integer *N* — The array is enlarged to include *N* more slots than it currently has.

a floating point number *F* — The array is changed to include *F* times the number of current slots.

a function or lambda expression *FN* — Upon hash overflow, *FN* is called with the hash array as its argument. If *FN* returns a number, that will become the size of the array. Otherwise, the new size defaults to 1.5 times its previous size. *FN* could be used to print a message, or perform some monitor function.

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

6.2 User-Specified Hashing Functions

In general terms, when a key is looked up in a hash array, it is converted to an integer, which is used to index into a linear array. If the key is not the same as the one found at that index, other indices are tried until it the desired key is found. The value stored with that key is then returned (from **GETHASH**) or replaced (from **PUTHASH**).

The important features of this algorithm, for purposes of customizing hash arrays, are (1) the "hashing function" used to convert a key to an integer; and (2) the comparison function used to compare the key found in the array with the key being looked up. In order for hash arrays to work correctly, any two objects which are equal according to the comparison function must "hash" to equal integers.

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

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

The user can override this default behavior for any hash array by specifying the functions used to compare keys and to "hash" a key to a number. This can be done by giving the *HASHBITSFN* and *EQUIVFN* arguments to **HASHARRAY** (page 6.1).

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

For an existing hash array, the function **HARRAYPROP** (page 6.2) can be used to examine the hashing and equivalence functions as the **HASHBITSFN** and **EQUIVFN** hash array properties. These properties are read-only for non-empty hash arrays, as it makes no sense to change the equivalence relationship once some keys have been hashed.

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

(STRINGHASHBITS *STRING*) [Function]

Hashes the string *STRING* into an integer that can be used as a **HASHBITSFN** for a hash array. Strings which are **STREQUAL** hash to the same integer.

Example:

(HASHARRAY *MINKEYS OVERFLOW* 'STRINGHASHBITS 'STREQUAL)

creates a hash array where you can use strings as hash keys.

[This page intentionally left blank]

[This page intentionally left blank]

7. NUMBERS AND ARITHMETIC FUNCTIONS

Numerical atoms, or simply numbers, do not have value cells, function definition cells, property lists, or explicit print names. There are four different types of numbers in Interlisp: small integers, large integers, bignums (arbitrary-size integers), and floating point numbers. Small integers are those integers that can be directly stored within a pointer value (implementation-dependent). Large integers and floating point numbers are full-word quantities that are stored by "boxing" the number (see below). Bignums are "boxed" as a series of words.

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

Note: Different implementations of Interlisp may use different boxing strategies. Thus, while lots of arithmetic operations *may* lead to garbage collections, this is not necessarily always the case.

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

(SMALLP *X*) [Function]

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

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

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

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

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

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

Returns *X*, if *X* is a number of any type (**FIXP** or **FLOATP**); **NIL** otherwise. Does *not* generate an error if *X* is not a number.

Note that if **(NUMBERP** *X***)** is true, then either **(FIXP** *X***)** or **(FLOATP** *X***)** is true.

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

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

Returns **T**, if *X* and *Y* are **EQ**, or equal numbers; **NIL** otherwise. Note that **EQ** may be used if *X* and *Y* are known to be *small* integers. **EQP** does not convert *X* and *Y* to integers, e.g., **(EQP 2000 2000.3) => NIL**, but it can be used to compare an integer and a floating point number, e.g., **(EQP 2000 2000.0) => T**. **EQP** does *not* generate an error if *X* or *Y* are not numbers.

Note: **EQP** can also be used to compare stack pointers (page 11.4) and compiled code objects (page 10.10).

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

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

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

For integer arithmetic: If *FLG* = **T**, an error occurs on division by zero. If *FLG* = **NIL** or **0**, integer division by zero returns zero. Integer overflow cannot occur, because small integers are converted to bignums (page 7.1).

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

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

7.1 Generic Arithmetic

The functions in this section are "generic" arithmetic functions. If any of the arguments are floating point numbers (page 7.11), they act exactly like floating point functions, and float all arguments, and return a floating point number as their value. Otherwise, they act like the integer functions (page 7.4). If given a non-numeric argument, they generate an error, **NON-NUMERIC ARG**.

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

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

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

$-X$

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

$X - Y$

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

$X_1 * X_2 * ... * X_N$

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

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

The results of division by zero and floating point overflow is determined by the function **OVERFLOW** (page 7.2).

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

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

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

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

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

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

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

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

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

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

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

(EQP *X* **0)**.

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

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

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

Returns the minimum of X_1, X_2, ..., X_N. **(MIN)** returns the value of **MAX.INTEGER** (page 7.5).

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

Returns the maximum of X_1, X_2, ..., X_N. **(MAX)** returns the value of **MIN.INTEGER** (page 7.5).

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

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

7.2 Integer Arithmetic

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

123Q
|o123 If an integer is followed by the letter **Q**, or proceeded by a vertical bar and the letter "o", the digits are interpreted an octal (base 8) integer.

|b10101 If an integer is proceeded by a vertical bar and the letter "b", the digits are interpreted as a binary (base 2) integer.

|x1A90 If an integer is proceeded by a vertical bar and the letter "x", the digits are interpreted as a hexadecimal (base 16) integer. The upper-case letters **A** though **F** are used as the digits after **9**.

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

Note that **77Q** and **63** both correspond to the same integers, and in fact are indistinguishable internally since no record is kept of the syntax used to create an integer. The function **RADIX** (page 25.13), sets the radix used to print integers.

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

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

MIN.SMALLP [Variable]

MAX.SMALLP [Variable]

The smallest/largest possible small integer.

MIN.FIXP [Variable]

MAX.FIXP [Variable]

The smallest/largest possible large integer.

MIN.INTEGER [Variable]

MAX.INTEGER [Variable]

The smallest/largest possible integers. For some algorithms, it is useful to have an integer that is larger than any other integer. Therefore, the values of **MAX.INTEGER** and **MIN.INTEGER** are two special bignums; the value of **MAX.INTEGER** is **GREATERP** than any other integer, and the value of **MIN.INTEGER** is **LESSP** than any other integer. Trying to do arithmetic using these special bignums, other than comparison, will cause an error.

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

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

Returns the sum $X_1 + X_2 + \ldots + X_N$. **(IPLUS) = 0**.

(IMINUS *X***)** [Function]

-*X*

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

X - *Y*

(ADD1 *X***)** [Function]

X + 1

(SUB1 *X***)** [Function]

X - 1

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

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

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

X / *Y* truncated. Examples:

(IQUOTIENT 3 2) => 1

(IQUOTIENT -3 2) => -1

If *Y* is zero, the result is determined by the function **OVERFLOW** (page 7.2).

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

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

(IREMAINDER 3 2) => 1

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

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

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

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

(ILESSP *X Y*) [Function]

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

(IGEQ *X Y*) [Function]

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

(ILEQ *X Y*) [Function]

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

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

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

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

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

(IEQP *X Y*) [Function]

Returns **T** if *X* and *Y* are **EQ** or equal integers; **NIL** otherwise. Note that **EQ** may be used if *X* and *Y* are known to be *small* integers. **IEQP** converts *X* and *Y* to integers, e.g., **(IEQP 2000 2000.3) => T**. Causes **NON-NUMERIC ARG** error if either *X* or *Y* are not numbers.

(FIX *N*) [Function]

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

Note: Since **FIX** is also a programmer's assistant command (page 13.12), typing **FIX** directly to Interlisp will not cause the function **FIX** to be called.

(FIXR *N*) [Function]

If *N* is an integer, returns *N*. Otherwise, converts *N* to an integer by rounding. For example, **(FIXR 2.3) => 2**, **(FIXR -1.7) => -2**, **(FIXR 3.5) => 4)**.

(GCD *N1 N2*) [Function]

Returns the greatest common divisor of *N1* and *N2*, e.g., **(GCD 72 64) = 8**.

7.3 Logical Arithmetic Functions

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

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

(LOGAND 7 5 6) => 4

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

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

(LOGOR 1 3 9) => 11

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

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

(LOGXOR 11 5) => 14

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

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

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

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

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

Warning: Be careful if using **RSH** to simulate division; **RSH**ing a negative number is not generally equivalent to dividing by a power of two.

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

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

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

Note: **LLSH** and **LRSH** are currently implemented using mod-2 ↑ 32 arithmetic. Passing a bignum to either of these will cause an error. **LRSH** of negative numbers will shift in 0s in the high bits.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Equivalent to

```
(LOADBYTE VAL
    (BYTEPOSITION BYTESPEC)
    (BYTESIZE BYTESPEC))
```

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

Equivalent to

```
(DEPOSITBYTE VAL
      (BYTEPOSITION BYTESPEC)
      (BYTESIZE BYTESPEC)
      N)
```

7.4 Floating Point Arithmetic

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

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

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

Floating point numbers are printed using the format control specified by the function **FLTFMT** (page 25.13). **FLTFMT** is initialized to **T**, or free format. For example, the above floating point numbers would be printed free format as:

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

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

PRINTNUM (page 25.15) permits greater controls on the printed appearance of floating point numbers, allowing such things as left-justification, suppression of trailing decimals, etc.

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

MIN.FLOAT [Variable]

The smallest possible floating point number.

MAX.FLOAT [Variable]

The largest possible floating point number.

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

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

$X_1 + X_2 + ... + X_N$

(FMINUS *X***)** [Function]

$-X$

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

$X - Y$

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

$X_1 * X_2 * ... * X_N$

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

X / Y.

The results of division by zero and floating point overflow is determined by the function **OVERFLOW** (page 7.2).

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

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

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

Example:

(FREMAINDER 7.5 2.3) => 0.6

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

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

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

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

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

Returns **T** if *N* and *M* are equal floating point numbers; **NIL** otherwise. **FEQP** converts *N* and *M* to floating point numbers.Causes **NON-NUMERIC ARG** error if either *N* or *M* are not numbers.

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

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

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

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

(FLOAT *X***)** [Function]

Converts *X* to a floating point number. Example:

(FLOAT 0) => 0.0

7.5 Other Arithmetic Functions

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

Returns *A* ↑ *N*. If *A* is an integer and *N* is a positive integer, returns an integer, e.g, **(EXPT 3 4) => 81**, otherwise returns a floating point number. If *A* is negative and *N* fractional, an error is generated, **ILLEGAL EXPONENTIATION**. If *N* is floating and either too large or too small, an error is generated, **VALUE OUT OF RANGE EXPT**.

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

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

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

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

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

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

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

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

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

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

Similar to **SIN**.

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

Similar to **SIN**.

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

X is a number between -1 and 1 (or an error is generated). The value of **ARCSIN** is a floating point number, and is in degrees unless *RADIANSFLG* = **T**. In other words, if **(ARCSIN** *X RADIANSFLG***)** = *Z* then **(SIN** *Z RADIANSFLG***)** = *X*. The range of the value of **ARCSIN** is -90 to +90 for degrees, -¯PI¯/2 to ¯PI¯/2 for radians.

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

Similar to **ARCSIN**. Range is 0 to 180, 0 to ¯PI¯.

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

Similar to **ARCSIN**. Range is 0 to 180, 0 to ¯PI¯.

(ARCTAN2 *Y X RADIANSFLG***)** [Function]

Computes **(ARCTAN (FQUOTIENT** *Y X***)** *RADIANSFLG***)**, and returns a corresponding value in the range -180 to 180 (or -¯PI¯ to ¯PI¯), i.e. the result is in the proper quadrant as determined by the signs of *X* and *Y*.

(RAND *LOWER UPPER***)** [Function]

Returns a pseudo-random number between *LOWER* and *UPPER* inclusive, i.e., **RAND** can be used to generate a sequence of random numbers. If both limits are integers, the value of **RAND** is an integer, otherwise it is a floating point number. The algorithm is completely deterministic, i.e., given the same initial state, **RAND** produces the same sequence of values. The internal state of **RAND** is initialized using the function **RANDSET** described below.

(RANDSET *X***)** [Function]

Returns the internal state of **RAND**. If *X* = **NIL**, just returns the current state. If *X* = **T**, **RAND** is initialized using the clocks, and **RANDSET** returns the new state. Otherwise, *X* is interpreted as a previous internal state, i.e., a value of **RANDSET**, and is used to reset **RAND**. For example,

```
←(SETQ OLDSTATE (RANDSET))
...
←(for X from 1 to 10 do (PRIN1 (RAND 1 10)))
2847592748NIL
←(RANDSET OLDSTATE)
...
←(for X from 1 to 10 do (PRIN1 (RAND 1 10)))
2847592748NIL
```

[This page intentionally left blank]

8. RECORD PACKAGE

The advantages of "data abstraction" have long been known: more readable code, fewer bugs, the ability to change the data structure without having to make major modifications to the program, etc. The record package encourages and facilitates this good programming practice by providing a uniform syntax for creating, accessing and storing data into many different types of data structures (arrays, list structures, association lists, etc.) as well as removing from the user the task of writing the various manipulation routines. The user declares (once) the data structures used by his programs, and thereafter indicates the manipulations of the data in a data-structure-independent manner. Using the declarations, the record package automatically computes the corresponding Interlisp expressions necessary to accomplish the indicated access/storage operations. If the data structure is changed by modifying the declarations, the programs automatically adjust to the new conventions.

The user describes the format of a data structure (record) by making a "record declaration" (see page 8.6). The record declaration is a description of the record, associating names with its various parts, or "fields". For example, the record declaration **(RECORD MSG (FROM TO . TEXT))** describes a data structure called **MSG**, which contains three fields: **FROM**, **TO**, and **TEXT**. The user can reference these fields by name, to retrieve their values or to store new values into them, by using the **FETCH** and **REPLACE** operators (page 8.2). The **CREATE** operator (page 8.3) is used for creating new instances of a record, and **TYPE?** (page 8.5) is used for testing whether an object is an instance of a particular record. (note: all record operators can be in either upper or lower case.)

Records may be implemented in a variety of different ways, as determined by the first element ("record type") of the record declaration. **RECORD** (used to specify elements and tails of a list structure) is just one of several record types currently implemented. The user can specify a property list format by using the record type **PROPRECORD**, or that fields are to be associated with parts of a data structure via a specified hash array by using the record type **HASHLINK**, or that an entirely new data type be allocated (as described on page 8.20) by using the record-type **DATATYPE**.

The record package is implemented through the DWIM/CLISP facilities, so it contains features such as spelling correction on field names, record types, etc. Record operations are translated using all CLISP declarations in effect (standard/fast/undoable); it is also possible to declare local record declarations that override global ones (see page 21.12).

The file package includes a **RECORDS** file package command for dumping record declarations (page 17.38), and **FILES?** and **CLEANUP** will inform the user about records that need to be dumped.

8.1 FETCH and REPLACE

The fields of a record are accessed and changed with the **FETCH** and **REPLACE** operators. If the record **MSG** has the record declaration **(RECORD MSG (FROM TO . TEXT))**, and **X** is a **MSG** data structure, **(fetch FROM of X)** will return the value of the **FROM** field of **X**, and **(replace FROM of X with Y)** will replace this field with the value of **Y**. In general, the value of a **REPLACE** operation is the same as the value stored into the field.

Note that the form **(fetch FROM of X)** implicitly states that **X** is an instance of the record **MSG**, or at least it should to be treated as such for this particular operation. In other words, the interpretation of **(fetch FROM of X)** never depends on the value of **X**. Therefore, if **X** is not a **MSG** record, this may produce incorrect results. The **TYPE?** record operation (page 8.5) may be used to test the types of objects.

If there is another record declaration, **(RECORD REPLY (TEXT . RESPONSE))**, then **(fetch TEXT of X)** is ambiguous, because **X** could be either a **MSG** or a **REPLY** record. In this case, an error will occur, **AMBIGUOUS RECORD FIELD**. To clarify this, **FETCH** and **REPLACE** can take a list for their "field" argument: **(fetch (MSG TEXT) of X)** will fetch the **TEXT** field of an **MSG** record. Note that if a field has an *identical* interpretation in two declarations, e.g. if the field **TEXT** occurred in the same location within the declarations of **MSG** and **REPLY**, then **(fetch TEXT of X)** would *not* be considered ambiguous.

An exception to this rule is that "user" record declarations take precedence over "system" record declarations, in cases where an unqualified field name would be considered ambiguous. System records are declared by including **(SYSTEM)** in the record declaration (see page 8.15). All of the records defined in the standard Interlisp-D system are defined as system records.

Another complication can occur if the fields of a record are themselves records. The fields of a record can be further broken down into sub-fields by a "subdeclaration" within the record declaration (see page 8.14). For example,

(RECORD NODE (POSITION . LABEL) (RECORD POSITION (XLOC . YLOC)))

permits the user to access the **POSITION** field with **(fetch POSITION of X)**, or its subfield **XLOC** with **(fetch XLOC of X)**.

The user may also elaborate a field by declaring that field name in a *separate* record declaration (as opposed to an embedded subdeclaration). For instance, the **TEXT** field in the **MSG** and **REPLY** records above may be subdivided with the seperate record declaration **(RECORD TEXT (HEADER . TXT))**. Fields of subfields (to any level of nested subfields) are accessed by specifying the "data path" as a list of record/field names, where there is some path from each record to the next in the list. For instance, **(fetch (MSG TEXT HEADER) of X)** indicates that **X** is to be treated as a **MSG** record, its **TEXT** field should be accessed, and *its* **HEADER** field should be accessed. Only as much of the data path as is necessary to disambiguate it needs to be specified. In this case, **(fetch (MSG HEADER) of X)** is sufficient. The record package interprets a data path by performing a tree search among all current record declarations for a path from each name to the next, considering first local declarations (page 21.13) and then global ones. The central point of separate declarations is that the (sub)record is *not* tied to another record (as with embedded declarations), and therefore can be used in many different contexts. If a data-path rather than a single field is ambiguous, (e.g., if there were yet another declaration **(RECORD TO (NAME . HEADER))** and the user specified **(fetch (MSG HEADER) of X)**), the error **AMBIGUOUS DATA PATH** is generated.

FETCH and **REPLACE** forms are translated using the CLISP declarations in effect (see page 21.12). **FFETCH** and **FREPLACE** are versions which insure fast CLISP declarations will be in effect, **/REPLACE** insures undoable declarations.

8.2 CREATE

Record operations can be applied to arbitrary structures, i.e., the user can explicitely creating a data structure (using **CONS**, etc), and then manipulate it with **FETCH** and **REPLACE**. However, to be consistant with the idea of data abstraction, new data should be created using the same declarations that define its data paths. This can be done with an expression of the form:

(create ***RECORD-NAME . ASSIGNMENTS*****)**

A **CREATE** expression translates into an appropriate Interlisp form using **CONS, LIST, PUTHASH, ARRAY**, etc., that creates the new datum with the various fields initialized to the appropriate values. *ASSIGNMENTS* is optional and may contain expressions of the following form:

FIELD-NAME ← *FORM* — Specifies initial value for *FIELD-NAME*.

USING *FORM* — Specifies that for all fields not explicitly given a value, the value of the corresponding field in *FORM* is to be used.

COPYING *FORM* — Similar to **USING** except the corresponding values are copied (with **COPYALL**).

REUSING *FORM* — Similar to **USING**, except that wherever possible, the corresponding *structure* in *FORM* is used.

SMASHING *FORM* — A new instance of the record is not created at all; rather, the value of *FORM* is used and smashed.

The record package goes to great pains to insure that the order of evaluation in the translation is the same as that given in the original **CREATE** expression if the side effects of one expression might affect the evaluation of another. For example, given the declaration **(RECORD CONS (CAR . CDR))**, the expression **(create CONS CDR←X CAR←Y)** will translate to **(CONS Y X)**, but **(create CONS CDR←(FOO) CAR←(FIE))** will translate to **((LAMBDA ($$1) (CONS (PROGN (SETQ $$1 (FOO)) (FIE)) $$1)))** because **FOO** might set some variables used by **FIE**.

Note that **(create** *RECORD* **REUSING** *FORM* **...)** does not itself do any destructive operations on the value of *FORM*. The distinction between **USING** and **REUSING** is that **(create** *RECORD* **reusing** *FORM* **...)** will incorporate as much as possible of the old data structure into the new one being created, while **(create** *RECORD* **using** *FORM* **...)** will create a completely new data structure, with only the contents of the fields re-used. For example, **REUSING** a **PROPRECORD** just **CONS**es the new property names and values onto the list, while **USING** copies the top level of the list. Another example of this distinction occurs when a field is elaborated by a subdeclaration (page 8.14): **USING** will create a new instance of the sub-record, while **REUSING** will use the old contents of the field (unless some field of the subdeclaration is assigned in the **CREATE** expression.)

If the value of a field is neither explicitly specified, nor implicitly specified via **USING**, **COPYING** or **REUSING**, the default value in the declaration is used, if any, otherwise **NIL**. (Note: For **BETWEEN** fields in **DATATYPE** records, N_1 is used; for other non-pointer fields zero is used.) For example, following **(RECORD A (B C D) D ← 3)**,

```
(create A B←T)
    ==> (LIST T NIL 3)

(create A B←T using X)
    ==> (LIST T (CADR X) (CADDR X))

(create A B←T copying X))
```

    ==> [LIST T (COPYALL (CADR X)) (COPYALL (CADDR X]

(create A B←T reusing X)

    ==> (CONS T (CDR X))

8.3 TYPE?

The record package allows the user to test if a given datum "looks like" an instance of a record. This can be done via an expression of the form

(type? *RECORD-NAME FORM***)**

TYPE? is mainly intended for records with a record type of **DATATYPE** or **TYPERECORD**. For **DATATYPE**s, the **TYPE?** check is exact; i.e. the **TYPE?** expression will return non-**NIL** only if the value of *FORM* is an instance of the record named by *RECORD-NAME*. For **TYPERECORD**s, the **TYPE?** expression will check that the value of *FORM* is a list beginning with *RECORD-NAME*. For **ARRAYRECORD**s, it checks that the value is an array of the correct size. For **PROPRECORD**s and **ASSOCRECORD**s, a **TYPE?** expression will make sure that the value of *FORM* is a property/association list with property names among the field-names of the declaration.

There is no built-in type test for records of type **ACCESSFNS**, **HASHLINK** or **RECORD**. Type tests can be defined for these kinds of records, or redefined for the other kinds, by including an expression of the form **(TYPE?** *COM***)** in the record declaration (see page 8.14). Attempting to execute a **TYPE?** expression for a record that has no type test causes an error, **TYPE? NOT IMPLEMENTED FOR THIS RECORD**.

8.4 WITH

Often one wants to write a complex expression that manipulates several fields of a single record. The **WITH** construct can make it easier to write such expressions by allowing one to refer to the fields of a record as if they were variables within a lexical scope:

(with *RECORD-NAME RECORD-INSTANCE* $FORM_1$ ... $FORM_N$**)**

RECORD-NAME is the name of a record, and *RECORD-INSTANCE* is an expression which evaluates to an instance of that record. The expressions $FORM_1$ ... $FORM_N$ are evaluated so that references to variables which are field-names of *RECORD-NAME* are implemented via **FETCH** and **SETQ**s of those variables are implemented via **REPLACE**.

For example, given

(RECORD RECN (FLD1 FLD2))
(SETQ INST (create RECN FLD1 ← 10 FLD2 ← 20))

Then the construct

(with RECN INST (SETQ FLD2 (PLUS FLD1 FLD2]

is equivalent to

(replace FLD2 of INST with (PLUS (fetch FLD1 of INST) (fetch FLD2 of INST]

Warning: **WITH** is implemented by doing simple substitutions in the body of the forms, without regard for how the record fields are used. This means, for example, if the record **FOO** is defined by **(RECORD FOO (POINTER1 POINTER2))**, then the form

(with FOO X (SELECTQ Y (POINTER1 POINTER1) NIL]

will be translated as

(SELECTQ Y ((CAR X) (CAR X)) NIL]

The user should be careful that record field names are not used except as variables in the **WITH** forms.

8.5 Record Declarations

A record is defined by evaluating a record declaration, which is an expression of the form:

(*RECORD-TYPE RECORD-NAME RECORD-FIELDS . RECORD-TAIL*)

RECORD-TYPE specifies the "type" of data being described by the record declaration, and thereby implicitly specifies how the corresponding access/storage operations are performed. The different record types are described below.

RECORD-NAME is a litatom used to identify the record declaration for creating instances of the record via **CREATE**, testing via **TYPE?**, and dumping to files via the **RECORDS** file package command (page 17.38). **DATATYPE** and **TYPERECORD** declarations also use *RECORD-NAME* to identify the data structure (as described below).

RECORD-FIELDS describes the structure of the record. Its exact interpretation varies with *RECORD-TYPE*. For most record types it defines the names of the fields within the record that can be accessed with **FETCH** and **REPLACE**.

RECORD-TAIL is an optional list that can be used to specify default values for record fields, special **CREATE** and **TYPE?** forms, and subdeclarations (described below).

Normally, record declaration forms are typed in to the top-level executive or read from a file, and they define the structure of the record globally. Local record declarations within the context of a function are defined by including a record declaration form in the CLISP declaration for the function, rather than evaluating the expression itself (see page 21.13).

Note: Although record declarations are evaluatable forms, and thus can be included in functions, changing a record declaration dynamically (at run-time) is not recommended. When a **FETCH** or **REPLACE** operation is interpreted, and the record declaration has changed, the form has to be re-translated. If a function containing **FETCH** or **REPLACE** operations has been compiled, it may be necessary to re-compile. For applications which need to change record declarations dynamically, users should consider using more flexible data structures, such as association lists or property lists.

8.5.1 Record Types

Records can be used to describe a large variety of data objects, that are manipulated in different ways. The *RECORD-TYPE* field of the record declaration specifies how the data object is created, and how the various record fields are accessed. Depending on the record type, the record fields may be stored in a list, or in an array, or on the property list of a litatom. The following record types are defined:

RECORD [Record Type]

The **RECORD** record type is used to describe list structures. *RECORD-FIELDS* is interpreted as a list structure whose non-**NIL** literal atoms are taken as field-names to be associated with the corresponding elements and tails of a list structure. For example, with the record declaration **(RECORD MSG (FROM TO . TEXT))**, **(fetch FROM of X)** translates as **(CAR X)**.

NIL can be used as a place marker to fill an unnamed field, e.g., **(A NIL B)** describes a three element list, with **B** corresponding to the third element. A number may be used to indicate a sequence of **NIL**s, e.g. **(A 4 B)** is interpreted as **(A NIL NIL NIL NIL B)**.

TYPERECORD [Record Type]

The **TYPERECORD** record type is similar to **RECORD**, except that the record name is added to the front of the list structure to signify what "type" of record it is. This type field is used by the record package in the translation of **TYPE?** expressions. **CREATE** will insert an extra field containing *RECORD-NAME* at the beginning of the structure, and the translation of the access and storage functions will take this extra field into account. For example, for **(TYPERECORD MSG (FROM TO . TEXT))**, **(fetch FROM of X)** translates as **(CADR X)**, not **(CAR X)**.

ASSOCRECORD [Record Type]

The **ASSOCRECORD** record type is used to describe list structures where the fields are stored in association list format:

(($FIELDNAME_1$. $VALUE_1$) ($FIELDNAME_2$. $VALUE_2$) ...)

RECORD-FIELDS is a list of literal atoms, interpreted as the permissable list of field names in the association list. Accessing is performed with **ASSOC** (or **FASSOC**, depending on current **CLISP** declarations, see page 21.12), storing with **PUTASSOC**.

PROPRECORD [Record Type]

The **PROPRECORD** record type is used to describe list structures where the fields are stored in property list format:

($FIELDNAME_1$ $VALUE_1$ $FIELDNAME_2$ $VALUE_2$ ...)

RECORD-FIELDS is a list of literal atoms, interpreted as the permissable list of field names in the property list. Accessing is performed with **LISTGET**, storing with **LISTPUT**.

Both **ASSOCRECORD** and **PROPRECORD** are useful for defining data structures in which it is often the case that many of the fields are **NIL**. A **CREATE** expression for these record types only stores those fields which are non-**NIL**. Note, however, that with the record declaration **(PROPRECORD FIE (H I J))** the expression **(create FIE)** would still construct **(H NIL)**, since a later operation of **(replace J of X with Y)** could not possibly change the instance of the record if it were **NIL**.

ARRAYRECORD [Record Type]

The **ARRAYRECORD** record type is used to describe arrays. *RECORD-FIELDS* is interpreted as a list of field names that are associated with the corresponding elements of an array. **NIL** can be used as a place marker for an unnamed field (element). Positive integers can be used as abbreviation for the corresponding number of **NIL**s. For example, **(ARRAYRECORD (ORG DEST NIL ID 3 TEXT))** describes an eight element array, with **ORG** corresponding to the first element, **ID** to the fourth, and **TEXT** to the eighth.

Note that **ARRAYRECORD** only creates arrays of pointers. Other kinds of arrays must be implemented by the user with the **ACCESSFNS** record type (page 8.12).

HASHLINK [Record Type]

The **HASHLINK** record type can be used with any type of data object: it specifies that the value of a single field can be accessed by hashing the data object in a given hash array. Since the **HASHLINK** record type describes an accessing method, rather than a data structure, the **CREATE** expression is meaningless for **HASHLINK** records.

RECORD-FIELDS is either an atom *FIELD-NAME*, or a list (*FIELD-NAME HARRAYNAME HARRAYSIZE*). *HARRAYNAME* is a variable whose value is the hash array to be used; if not given, **SYSHASHARRAY** is used. If the value of the variable *HARRAYNAME* is not a hash array (at the time of the record declaration), it will be set to a new hash array with a size of *HARRAYSIZE*. *HARRAYSIZE* defaults to 100.

The **HASHLINK** record type is useful as a subdeclaration to other records to add additional fields to already existing data structures (see page 8.14). For example, suppose that **FOO** is a record declared with **(RECORD FOO (A B C))**. To add an aditional field **BAR**, without modifying the already existing data strutures, redeclare **FOO** with:

(RECORD FOO (A B C) (HASHLINK FOO (BAR BARHARRAY)))

Now, **(fetch BAR of X)** will translate into **(GETHASH X BARHARRAY)**, hashing off the existing list **X**.

ATOMRECORD [Record Type]

The **ATOMRECORD** record type is used to describe property lists of litatoms. *RECORD-FIELDS* is a list of property names. Accessing is performed with **GETPROP**, storing with **PUTPROP**. The **CREATE** expression is not initially defined for **ATOMRECORD** records.

DATATYPE [Record Type]

The **DATATYPE** record type is used to define a new user data type with type name *RECORD-NAME* (by calling **DECLAREDATATYPE**, page 8.21). Unlike other record types, the records of a **DATATYPE** declaration are represented with a completely new Interlisp type, and not in terms of other existing types.

RECORD-FIELDS is interpreted as a list of field specifications, where each specification is either a list (*FIELDNAME FIELDTYPE*), or an atom *FIELDNAME*. If *FIELDTYPE* is omitted, it defaults to **POINTER**. Possible values for *FIELDTYPE* are:

POINTER Field contains a pointer to any arbitrary Interlisp object.

INTEGER

FIXP Field contains a signed integer. Note that an **INTEGER** field is not capable of holding everything that satisfies **FIXP**, such as bignums (page 7.1).

FLOATING

FLOATP Field contains a floating point number.

SIGNEDWORD Field contains a 16-bit signed integer.

FLAG Field is a one bit field that "contains" **T** or **NIL**.

BITS *N* Field contains an *N*-bit unsigned integer.

BYTE Equivalent to **BITS 8**.

WORD Equivalent to **BITS 16**.

XPOINTER Field contains a pointer like **POINTER**, except that the field is *not* reference counted by the Interlisp-D garbage collector. **XPOINTER** fields are useful for implementing back-pointers in structures that would be circular and not otherwise collected by the reference-counting garbage collector.

Warning: **XPOINTER** fields should be used with great care. It is possible to damage the integrity of the storage allocation system by using pointers to objects that have been garbage collected. Code that uses **XPOINTER** fields should be sure that the objects pointed to have not been garbage collected. This can be done in two ways: The first is to maintain the object in a global structure, so that it is never garbage collected until explicitly deleted from the structure, at which point the program must invalidate all the **XPOINTER** fields of other objects pointing at it. The second is to declare the object as a **DATATYPE** beginning with a **POINTER** field that the program maintains as a pointer to an object of another type (e.g., the object containing the **XPOINTER** pointing back at it), and test that field for reasonableness whenever using the contents of the **XPOINTER** field.

For example, the declaration

```
(DATATYPE FOO
 ((FLG BITS 12)
  TEXT
  HEAD
  (DATE BITS 18)
  (PRIO FLOATP)
  (READ? FLAG)))
```

would define a data type **FOO** with two pointer fields, a floating point number, and fields for a 12 and 18 bit unsigned integers, and a flag (one bit). Fields are allocated in such a way as to optimize the storage used and not necessarily in the order specified. Generally, a **DATATYPE** record is much more storage compact than the corresponding **RECORD** structure would be; in addition, access is faster.

Since the user data type must be set up at *run*-time, the **RECORDS** file package command will dump a **DECLAREDATATYPE** expression as well as the **DATATYPE** declaration itself. If the record declaration is otherwise not needed at runtime, it can be kept out of the compiled file by using a **(DECLARE: DONTCOPY --)** expression (see page 17.40), but it is still necessary to ensure that the datatype is properly initialized. For this, one can use the **INITRECORDS** file package command (page 17.38), which will dump only the **DECLAREDATATYPE** expression.

Note: When defining a new data type, it is sometimes useful to call the function **DEFPRINT** (page 25.16) to specify how instances of the new data type should be printed. This can be specified in the record declaration by including an **INIT** record specification (page 8.14), e.g. **(DATATYPE QV.TYPE ... (INIT (DEFPRINT 'QV.TYPE (FUNCTION PRINT.QV.TYPE))))**.

Note: **DATATYPE** declarations cannot be used within local record declarations (page 21.13).

BLOCKRECORD [Record Type]

The **BLOCKRECORD** record type is used in low-level system programming to "overlay" an organized structure over an arbitrary piece of "unboxed" storage. *RECORD-FIELDS* is interpreted exactly as with a **DATATYPE** declaration, except that fields are *not* automatically rearranged to maximize storage efficiency. Like an **ACCESSFNS** record, a **BLOCKRECORD** does not have concrete instances; it merely provides a way of interpreting some existing block of storage. Thus, one cannot create an instance of a **BLOCKRECORD** (unless the declaration includes an explicit **CREATE** expression), nor is there a default **type?** expression for a **BLOCKRECORD**.

Warning: The programmer should exercise caution in using **BLOCKRECORD** declarations, as they enable one to write expressions that fetch and store arbitrary data in arbitrary locations, thereby evading the normal type system. Except in very specialized situations, a **BLOCKRECORD** should never contain **POINTER** or **XPOINTER** fields, nor be used to overlay an area of storage that contains pointers. Such use could compromise the garbage collector and storage allocation system. The programmer is responsible for ensuring that all **FETCH** and **REPLACE** expressions are performed only on suitable objects, as no type testing is performed.

A typical use for the **BLOCKRECORD** type in user code is to overlay a non-pointer portion of an existing **DATATYPE**. For this use, the **LOCF** macro is useful. **(LOCF (fetch *FIELD* of *DATUM*))** can be used to refer to the storage that begins at the first word that contains *FIELD* of *DATUM*. For example, to define a new kind of Ethernet packet (page 31.26), one could overlay the "body" portion of the **ETHERPACKET** datatype declaration as follows:

```
(ACCESSFNS MYPACKET
   ((MYBASE (LOCF (fetch (ETHERPACKET EPBODY) of DATUM))))
   (BLOCKRECORD MYBASE
      ((MYTYPE WORD)
       (MYLENGTH WORD)
       (MYSTATUS BYTE)
       (MYERRORCODE BYTE)
       (MYDATA INTEGER)))
   (TYPE? (type? ETHERPACKET DATUM)))
```

With this declaration in effect, the expression **(fetch MYLENGTH of PACKET)** would retrieve the second 16-bit field beyond the offset inside **PACKET** where the **EPBODY** field starts. For more examples, see the EtherRecords library package.

ACCESSFNS [Record Type]

The **ACCESSFNS** record type is used to define data structures with user-defined access functions. For each field name, the user specifies how it is to be accessed and set. This allows the use of the record package with arbitrary data structures, with complex access methods.

RECORD-FIELDS is interpreted as a list of elements of the form (*FIELD-NAME ACCESSDEF SETDEF*). *ACCESSDEF* should be a function of one argument, the datum, and will be used for accessing the value of the field. *SETDEF* should be a function of two arguments, the datum and the new value, and will be used for storing a new value in a field. *SETDEF* may be omitted, in which case, no storing operations are allowed.

ACCESSDEF and/or *SETDEF* may also be a form written in terms of variables **DATUM** and (in *SETDEF*) **NEWVALUE**. For example, given the declaration

```
[ACCESSFNS FOO
  ((FIRSTCHAR (NTHCHAR DATUM 1)
        (RPLSTRING DATUM 1 NEWVALUE))
  (RESTCHARS (SUBSTRING DATUM 2]
```

(replace (FOO FIRSTCHAR) of X with Y) would translate to **(RPLSTRING X 1 Y)**. Since no *SETDEF* is given for the **RESTCHARS** field, attempting to perform **(replace (FOO RESTCHARS) of X with Y)** would generate an error, **REPLACE UNDEFINED FOR FIELD**. Note that **ACCESSFNS** do not have a **CREATE** definition. However, the user may supply one in the defaults or subdeclarations of the declaration, as described below.

Attempting to **CREATE** an **ACCESSFNS** record without specifying a create definition will cause an error **CREATE NOT DEFINED FOR THIS RECORD**.

ACCESSDEF and *SETDEF* can also be a property list which specify **FAST, STANDARD** and **UNDOABLE** versions of the **ACCESSFNS** forms, e.g.

```
[ACCESSFNS LITATOM
 ((DEF (STANDARD GETD FAST FGETD)
    (STANDARD PUTD UNDOABLE /PUTD]
```

means if **FAST** declaration is in effect, use **FGETD** for fetching, if **UNDOABLE**, use **/PUTD** for saving (see CLISP declarations, page 21.12).

Note: *SETDEF* forms should be written so that they return the new value, to be consistant with **REPLACE** operations for other record types. The **REPLACE** record operator does not enforce this, though.

The **ACCESSFNS** facility allows the use of data structures not specified by one of the built-in record types. For example, one possible representation of a data structure is to store the fields in *parallel* arrays, especially if the number of instances required is known, and they do not need to be garbage collected Thus, to implement a data structure called **LINK** with two fields **FROM** and **TO**, one would have two arrays **FROMARRAY** and **TOARRAY**. The representation of an "instance" of the record would be an integer which is used to index into the arrays. This can be accomplished with the declaration:

```
[ACCESSFNS LINK
 ((FROM (ELT FROMARRAY DATUM)
     (SETA FROMARRAY DATUM NEWVALUE))
  (TO (ELT TOARRAY DATUM)
     (SETA TOARRAY DATUM NEWVALUE)))
  (CREATE (PROG1 (SETQ LINKCNT (ADD1 LINKCNT))
          (SETA FROMARRAY LINKCNT FROM)
          (SETA TOARRAY LINKCNT TO)))
  (INIT (PROGN
       (SETQ FROMARRAY (ARRAY 100))
       (SETQ TOARRAY (ARRAY 100))
       (SETQ LINKCNT 0)]
```

To create a new **LINK**, a counter is incremented and the new elements stored. (Note: The **CREATE** form given the declaration probably should include a test for overflow.)

8.5.2 Optional Record Specifications

After the *RECORD-FIELDS* item in a record declaration expression there can be an arbitrary number of additional expressions in *RECORD-TAIL*. These expressions can be used to specify default values for record fields, special **CREATE** and **TYPE?** forms, and subdeclarations. The following expressions are permitted:

FIELD-NAME ← *FORM*
: Allows the user to specify within the record declaration the default value to be stored in *FIELD-NAME* by a **CREATE** (if no value is given within the **CREATE** expression itself). Note that *FORM* is evaluated at **CREATE** time, not when the declaration is made.

(CREATE *FORM***)**
: Defines the manner in which **CREATE** of this record should be performed. This provides a way of specifying how **ACCESSFNS** should be created or overriding the usual definition of **CREATE**. If *FORM* contains the field-names of the declaration as variables, the forms given in the **CREATE** operation will be substituted in. If the word **DATUM** appears in the create form, the *original* **CREATE** definition is inserted. This effectively allows the user to "advise" the create.

Note: **(CREATE** *FORM***)** may also be specified as "*RECORD-NAME* ← *FORM*".

(INIT *FORM***)**
: Specifies that *FORM* should be evaluated when the record is declared. *FORM* will also be dumped by the **INITRECORDS** file package command (page 17.38).

For example, see the example of an **ACCESSFNS** record declaration above. In this example, **FROMARRAY** and **TOARRAY** are initialized with an **INIT** form.

(TYPE? *FORM***)**
: Defines the manner in which **TYPE?** expressions are to be translated. *FORM* may either be an expression in terms of **DATUM** or a function of one argument.

(SUBRECORD *NAME* . *DEFAULTS***)**
: *NAME* must be a field that appears in the current declaration and the name of another record. This says that, for the purposes of translating **CREATE** expressions, substitute the top-level declaration of *NAME* for the **SUBRECORD** form, adding on any defaults specified.

For example: Given **(RECORD B (E F G))**, **(RECORD A (B C D) (SUBRECORD B))** would be treated like **(RECORD A (B C D) (RECORD B (E F G)))** for the purposes of translating **CREATE** expressions.

a subdeclaration
: If a record declaration expression occurs among the record specifications of another record declaration, it is known as a "subdeclaration." Subdeclarations are used to declare that fields of a record are to be interpreted as another type of record, or that the record data object is to be interpreted in more than one way.

The *RECORD-NAME* of a subdeclaration must be either the *RECORD-NAME* of its immediately superior declaration or one of the superior's field-names. Instead of identifying the declaration as with top level declarations, the record-name of a subdeclaration identifies the parent field or record that is being described by the subdeclaration. Subdeclarations can be nested to an arbitrary depth.

Giving a subdeclaration **(RECORD** $NAME_1$ $NAME_2$**)** is a simple way of defining a *synonym* for the field $NAME_1$.

It is possible for a given field to have more than one subdeclaration. For example, in

(RECORD FOO (A B) (RECORD A (C D)) (RECORD A (Q R)))

(Q R) and **(C D)** are "overlayed," i.e. **(fetch Q of X)** and **(fetch C of X)** would be equivalent. In such cases, the *first* subdeclaration is the one used by **CREATE**.

(SYNONYM *FIELD* (SYN_1 ... SYN_N)**)**

FIELD must be a field that appears in the current declaration. This defines SYN_1 ... SYN_N all as synonyms of *FIELD*. If there is only one synonym, this can be written as **(SYNONYM** *FIELD SYN***)**.

(SYSTEM)

If **(SYSTEM)** is included in a record declaration, this indicates that the record is a "system" record rather than a "user" record. The only distinction between the two types of records is that "user" record declarations take precedence over "system" record declarations, in cases where an unqualified field name would be considered ambiguous. All of the records defined in the standard Interlisp-D system are defined as system records.

8.6 Defining New Record Types

In addition to the built-in record types, users can declare their own record types by performing the following steps:

(1) Add the new record-type to the value of **CLISPRECORDTYPES**;

(2) Perform **(MOVD 'RECORD** *RECORD-TYPE***)**, i.e. give the record-type the same definition as that of the function **RECORD**;

(3) Put the name of a function which will return the translation on the property list of *RECORD-TYPE*, as the value of the property **USERRECORDTYPE**. Whenever a record declaration of type *RECORD-TYPE* is encountered, this function will be passed the record declaration as its argument, and should return a *new* record declaration which the record package will then use in its place.

8.7 Record Manipulation Functions

(EDITREC *NAME* COM_1 ... COM_N**)** [NLambda NoSpread Function]

EDITREC calls the editor on a copy of all declarations in which *NAME* is the record name or a field name. On exit, it redeclares those that have changed and undeclares any that have been deleted. If *NAME* is **NIL**, *all* declarations are edited.

COM_1 ... COM_N are (optional) edit commands.

When the user redeclares a global record, the translations of all expressions involving that record or any of its fields are automatically deleted from **CLISPARRAY**, and thus will be recomputed using the new information. If the user changes a *local* record declaration (page 21.13), or changes some other CLISP declaration (page 21.12), e.g., **STANDARD** to **FAST**, and wishes the new information to affect record expressions already translated, he must make sure the corresponding translations are removed, usually either by **CLISPIFY**ing or using the **DW** edit macro.

(RECLOOK *RECNAME* — — — —**)** [Function]

Returns the entire declaration for the record named *RECNAME*; **NIL** if there is no record declaration with name *RECNAME*. Note that the record package maintains internal state about current record declarations, so performing destructive operations (e.g. **NCONC**) on the value of **RECLOOK** may leave the record package in an inconsistent state. To change a record declaration, use **EDITREC**.

(FIELDLOOK *FIELDNAME***)** [Function]

Returns the list of declarations in which *FIELDNAME* is the name of a field.

(RECORDFIELDNAMES *RECORDNAME* —**)** [Function]

Returns the list of fields declared in record *RECORDNAME*. *RECORDNAME* may either be a name or an entire declaration.

(RECORDACCESS *FIELD DATUM DEC TYPE NEWVALUE***)** [Function]

TYPE is one of **FETCH**, **REPLACE**, **FFETCH**, **FREPLACE**, **/REPLACE** or their lowercase equivalents. *TYPE* = **NIL** means **FETCH**. If *TYPE* corresponds to a fetch operation, i.e. is **FETCH**, or **FFETCH**, **RECORDACCESS** performs **(***TYPE FIELD* **of** *DATUM***)**. If *TYPE* corresponds to a replace, **RECORDACCESS** performs **(***TYPE FIELD* **of** *DATUM* **with** *NEWVALUE***)**. *DEC* is an optional declaration; if given, *FIELD* is interpreted as a field name of that declaration.

Note that **RECORDACCESS** is relatively inefficient, although it is better than constructing the equivalent form and performing an **EVAL**.

(RECORDACCESSFORM *FIELD DATUM TYPE NEWVALUE***)** [Function]

Returns the form that would be compiled as a result of a record access. *TYPE* is one of **FETCH**, **REPLACE**, **FFETCH**, **FREPLACE**, **/REPLACE** or their lowercase equivalents. *TYPE* = **NIL** means **FETCH**.

8.8 Changetran

A very common programming construction consists of assigning a new value to some datum that is a function of the current value of that datum. Some examples of such read-modify-write sequences include:

Incrementing a counter:

(SETQ X (IPLUS X 1))

Pushing an item on the front of a list:

(SETQ X (CONS Y X))

Popping an item off a list:

(PROG1 (CAR X) (SETQ X (CDR X)))

It is easier to express such computations when the datum in question is a simple variable as above than when it is an element of some larger data structure. For example, if the datum to be modified was **(CAR X)**, the above examples would be:

(CAR (RPLACA X (IPLUS (CAR X) 1)))

(CAR (RPLACA X (CONS Y (CAR X)))

(PROG1 (CAAR X) (RPLACA X (CDAR X)))

and if the datum was an element in an array, **(ELT A N)**, the examples would be:

(SETA A N (IPLUS (ELT A N) 1)))

(SETA A N (CONS Y (ELT A N))))

(PROG1 (CAR (ELT A N)) (SETA A N (CDR (ELT A N))))

The difficulty in expressing (and reading) modification idioms is in part due to the well-known asymmetry of setting versus accessing operations on structures: **RPLACA** is used to set what **CAR** would return, **SETA** corresponds to **ELT**, and so on.

The "Changetran" facility is designed to provide a more satisfactory notation in which to express certain common (but user-extensible) structure modification operations. Changetran defines a set of CLISP words that encode the kind of modification that is to take place, e.g. pushing on a list, adding to a number, etc. More important, the expression that indicates the datum whose value is to be modified needs to be stated only once. Thus, the "change word" **ADD** is used to increase the value of a datum by the sum of a set of numbers. Its arguments are an expression denoting the datum, and a set of items to be added to its current value. The datum expression must be a variable or an accessing expression (envolving **FETCH**, **CAR**, **LAST**, **ELT**, etc) that can be translated to the appropriate setting expression.

For example, **(ADD (CADDR X) (FOO))** is equivalent to:

```
(CAR (RPLACA (CDDR X)
       (PLUS (FOO) (CADDR X)))
```

If the datum expression is a complicated form involving subsidiary function calls, such as **(ELT (FOO X) (FIE Y))**, Changetran goes to some lengths to make sure that those subsidiary functions are evaluated only once (it binds local variables to save the results), even though they logically appear in both the setting and accessing parts of the translation. Thus, in thinking about both efficiency and possible side effects, the user can rely on the fact that the forms will be evaluated only as often as they appear in the expression.

For **ADD** and all other changewords, the lower-case version (**add**, etc.) may also be specified. Like other CLISP words, change words are translated using all CLISP declarations in effect (see page 21.12).

The following is a list of those change words recognized by Changetran. Except for **POP**, the value of all built-in changeword forms is defined to be the new value of the datum.

(ADD *DATUM ITEM$_1$ ITEM$_2$* **...)** [Change Word]

Adds the specified items to the current value of the datum, stores the result back in the datum location. The translation will use **IPLUS**, **PLUS**, or **FPLUS** according to the CLISP declarations in effect (see page 21.12).

(PUSH *DATUM ITEM$_1$ ITEM$_2$* **...)** [Change Word]

CONSes the items onto the front of the current value of the datum, and stores the result back in the datum location. For example, **(PUSH X A B)** would translate as **(SETQ X (CONS A (CONS B X)))**.

(PUSHNEW *DATUM ITEM***)** [Change Word]

Like **PUSH** (with only one item) except that the item is not added if it is already **FMEMB** of the datum's value.

Note that, whereas **(CAR (PUSH X 'FOO))** will always be **FOO**, **(CAR (PUSHNEW X 'FOO))** might be something else if **FOO** already existed in the middle of the list.

(PUSHLIST *DATUM ITEM*$_1$ *ITEM*$_2$ ...**)** [Change Word]

Similar to **PUSH**, except that the items are **APPEND**ed in front of the current value of the datum. For example, **(PUSHLIST X A B)** would translate as **(SETQ X (APPEND A B X))**.

(POP *DATUM***)** [Change Word]

Returns **CAR** of the current value of the datum after storing its **CDR** into the datum. The current value is computed only once even though it is referenced twice. Note that this is the only built-in changeword for which the value of the form is not the new value of the datum.

(SWAP *DATUM*$_1$ *DATUM*$_2$**)** [Change Word]

Sets *DATUM*$_1$ to *DATUM*$_2$ and vice versa.

(CHANGE *DATUM FORM***)** [Change Word]

This is the most flexible of all change words, since it enables the user to provide an arbitrary form describing what the new value should be, but it still highlights the fact that structure modification is to occur, and still enables the datum expression to appear only once. **CHANGE** sets *DATUM* to the value of *FORM**, where *FORM** is constructed from *FORM* by substituting the datum expression for every occurrence of the litatom **DATUM**. For example, **(CHANGE (CAR X) (ITIMES DATUM 5))** translates as **(CAR (RPLACA X (ITIMES (CAR X) 5)))**.

CHANGE is useful for expressing modifications that are not built-in and are not sufficiently common to justify defining a user-changeword. As for other changeword expressions, the user need not repeat the datum-expression and need not worry about multiple evaluation of the accessing form.

It is possible for the user to define new change words. To define a change word, say **sub**, that subtracts items from the current value of the datum, the user must put the property **CLISPWORD**, value **(CHANGETRAN . sub)** on both the upper and lower-case versions of **sub**:

(PUTPROP 'SUB 'CLISPWORD '(CHANGETRAN . sub))
(PUTPROP 'sub 'CLISPWORD '(CHANGETRAN . sub))

Then, the user must put (on the *lower*-case version of **sub** only) the property **CHANGEWORD**, with value *FN*. *FN* is a function that will be applied to a single argument, the whole **sub** form, and must return a form that Changetran can translate into an appropriate expression. This form should be a list structure with the atom **DATUM** used whenever the user wants an accessing expression for the current value of the datum to appear. The form **(DATUM← *FORM*)** (note that **DATUM←** is a single atom) should occur once in the expression; this specifies that an appropriate storing expression into the datum should occur at that point. For example, **sub** could be defined with:

```
(PUTPROP 'sub 'CHANGEWORD
     '(LAMBDA (FORM)
       (LIST 'DATUM←
          (LIST 'IDIFFERENCE
             'DATUM
             (CONS 'IPLUS (CDDR FORM))))))
```

If the expression **(sub (CAR X) A B)** were encountered, the arguments to **SUB** would first be dwimified, and then the **CHANGEWORD** function would be passed the list **(sub (CAR X) A B)**, and return **(DATUM← (IDIFFERENCE DATUM (IPLUS A B)))**, which Changetran would convert to **(CAR (RPLACA X (IDIFFERENCE (CAR X) (IPLUS A B))))**.

Note: The **sub** changeword as defined above will always use **IDIFFERENCE** and **IPLUS**; **add** uses the correct addition operation depending on the current CLISP declarations (see page 21.12).

8.9 Built-In and User Data Types

Interlisp is a system for the manipulation of various kinds of data; it provides a large set of built-in data types, which may be used to represent a variety of abstract objects, and the user can also define additional "user data types" which can be manipulated exactly like built-in data types.

Each data type in Interlisp has an associated "type name," a litatom. Some of the type names of built-in data types are: **LITATOM**, **LISTP**, **STRINGP**, **ARRAYP**, **STACKP**, **SMALLP**, **FIXP**, and **FLOATP**. For user data types, the type name is specified when the data type is created.

(DATATYPES —) [Function]

Returns a list of all type names currently defined.

(USERDATATYPES) [Function]

Returns list of names of currently declared user data types.

(TYPENAME *DATUM*) [Function]

Returns the type name for the data type of *DATUM*.

(TYPENAMEP *DATUM TYPE***) [Function]**

Returns **T** if *DATUM* is an object with type name equal to *TYPE*, otherwise **NIL**.

Note: **TYPENAME** and **TYPENAMEP** distinguish the logical data types **ARRAYP, CCODEP** and **HARRAYP**, even though they may be implemented as **ARRAYPs** in some Interlisp implementations.

In addition to built-in data-types such as atoms, lists, arrays, etc., Interlisp provides a way of defining completely *new* classes of objects, with a fixed number of fields determined by the definition of the data type. In order to define a new class of objects, the user must supply a name for the new data type and specifications for each of its fields. Each field may contain either a pointer (i.e., any arbitrary Interlisp datum), an integer, a floating point number, or an *N*-bit integer.

Note: The most convenient way to define new user data types is via **DATATYPE** record declarations (page 8.9) which call the following functions.

(DECLAREDATATYPE *TYPENAME FIELDSPECS* **— —) [Function]**

Defines a new user data type, with the name *TYPENAME*. *FIELDSPECS* is a list of "field specifications." Each field specification may be one of the following:

POINTER Field may contain any Interlisp datum.

FIXP Field contains an integer.

FLOATP Field contains a floating point number.

(BITS *N***)** Field contains a non-negative integer less than 2^N.

BYTE Equivalent to **(BITS 8)**.

WORD Equivalent to **(BITS 16)**.

SIGNEDWORD Field contains a 16 bit signed integer.

DECLAREDATATYPE returns a list of "field descriptors," one for each element of *FIELDSPECS*. A field descriptor contains information about where within the datum the field is actually stored.

If *FIELDSPECS* is **NIL**, *TYPENAME* is "undeclared." If *TYPENAME* is already declared as a data type, it is undeclared, and then re-declared with the new *FIELDSPECS*. An instance of a data type that has been undeclared has a type name of ****DEALLOC****.

(FETCHFIELD *DESCRIPTOR DATUM***) [Function]**

Returns the contents of the field described by *DESCRIPTOR* from *DATUM*. *DESCRIPTOR* must be a "field descriptor" as returned by **DECLAREDATATYPE** or **GETDESCRIPTORS**. If *DATUM* is not an instance of the datatype of which *DESCRIPTOR* is a descriptor, causes error **DATUM OF INCORRECT TYPE**.

(REPLACEFIELD *DESCRIPTOR DATUM NEWVALUE***)** [Function]

Store *NEWVALUE* into the field of *DATUM* described by *DESCRIPTOR*. *DESCRIPTOR* must be a field descriptor as returned by **DECLAREDATATYPE**. If *DATUM* is not an instance of the datatype of which *DESCRIPTOR* is a descriptor, causes error **DATUM OF INCORRECT TYPE**. Value is *NEWVALUE*.

(NCREATE *TYPE OLDOBJ***)** [Function]

Creates and returns a new instance of datatype *TYPE*.

If *OLDOBJ* is also a datum of datatype *TYPE*, the fields of the new object are initialized to the values of the corresponding fields in *OLDOBJ*.

NCREATE will not work for built-in datatypes, such as **ARRAYP**, **STRINGP**, etc. If *TYPE* is not the type name of a previously declared *user* data type, generates an error, **ILLEGAL DATA TYPE**.

(GETFIELDSPECS *TYPENAME***)** [Function]

Returns a list which is **EQUAL** to the *FIELDSPECS* argument given to **DECLAREDATATYPE** for *TYPENAME*; if *TYPENAME* is not a currently declared data-type, returns **NIL**.

(GETDESCRIPTORS *TYPENAME***)** [Function]

Returns a list of field descriptors, **EQUAL** to the *value* of **DECLAREDATATYPE** for *TYPENAME*. If *TYPENAME* is not an atom, **(TYPENAME** *TYPENAME***)** is used.

Note that the user can define how user data types are to be printed via **DEFPRINT** (page 25.16), how they are to be evaluated by the interpreter via **DEFEVAL** (page 10.13), and how they are to be compiled by the compiler via **COMPILETYPELST** (page 18.11).

[This page intentionally left blank]

9. CONDITIONALS AND ITERATIVE STATEMENTS

In order to do any but the simplest computations, it is necessary to test values and execute expressions conditionally, and to execute a series of expressions. Interlisp supplies a large number of predicates, conditional functions, and control functions. Also, there is a complex "iterative statement" facility which allows the user to easily create complex loops and iterative constructs (page 9.9).

9.1 Data Type Predicates

Interlisp provides separate functions for testing whether objects are of certain commonly-used types:

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

Returns **T** if *X* is a litatom (see page 2.1) **NIL** otherwise. Note that a number is not a litatom.

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

Returns *X* if *X* is a small integer; **NIL** otherwise. (Note that the range of small integers is implementation-dependent. See page 7.1.)

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

Returns *X* if *X* is a small or large integer; **NIL** otherwise.

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

Returns *X* if *X* is a floating point number; **NIL** otherwise.

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

Returns *X* if *X* is a number of any type (**FIXP** or **FLOATP**), **NIL** otherwise.

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

Returns **T** if *X* is an atom (i.e. a litatom or a number); **NIL** otherwise.

Warning: **(ATOM *X*)** is **NIL** if *X* is an array, string, etc. In many dialects of Lisp, the function **ATOM** is defined equivalent to the Interlisp function **NLISTP**.

(LISTP *X*) [Function]

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

(NLISTP *X*) [Function]

(NOT (LISTP X)). Returns **T** if *X* is not a list cell, **NIL** otherwise.

(STRINGP *X*) [Function]

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

(ARRAYP *X*) [Function]

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

Note: In some implementations of Interlisp (but not Interlisp-D), **ARRAYP** may also return *X* if it is of type **CCODEP** or **HARRAYP**.

(HARRAYP *X*) [Function]

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

Note that **HARRAYP** returns **NIL** if *X* is a list whose **CAR** is an **HARRAYP**, even though this is accepted by the hash array functions.

Note: The empty list, **()** or **NIL**, is considered to be a litatom, rather than a list. Therefore, **(LITATOM NIL) = (ATOM NIL) = T** and **(LISTP NIL) = NIL**. Care should be taken when using these functions if the object may be the empty list **NIL**.

9.2 Equality Predicates

A common operation when dealing with data objects is to test whether two objects are equal. In some cases, such as when comparing two small integers, equality can be easily determined. However, sometimes there is more than one type of equality. For instance, given two lists, one can ask whether they are exactly the same object, or whether they are two distinct lists which contain the same elements. Confusion between these two types of equality is often the source of program errors. Interlisp supplies an extensive set of functions for testing equality:

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

Returns **T** if *X* and *Y* are identical pointers; **NIL** otherwise. **EQ** should not be used to compare two numbers, unless they are small integers; use **EQP** instead.

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

(NOT (EQ *X Y***))**

(NULL *X***)** [Function]

(NOT *X***)** [Function]

(EQ *X* **NIL)**

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

Returns **T** if *X* and *Y* are **EQ**, or if *X* and *Y* are numbers and are equal in value; **NIL** otherwise. For more discussion of **EQP** and other number functions, see page 7.1.

Note: **EQP** also can be used to compare stack pointers (page 11.4) and compiled code (page 10.10).

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

EQUAL returns **T** if *X* and *Y* are (1) **EQ**; or (2) **EQP**, i.e., numbers with equal value; or (3) **STREQUAL**, i.e., strings containing the same sequence of characters; or (4) lists and **CAR** of *X* is **EQUAL** to **CAR** of *Y*, and **CDR** of *X* is **EQUAL** to **CDR** of *Y*. **EQUAL** returns **NIL** otherwise. Note that **EQUAL** can be significantly slower than **EQ**.

A loose description of **EQUAL** might be to say that *X* and *Y* are **EQUAL** if they print out the same way.

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

Like **EQUAL**, except it descends into the contents of arrays, hash arrays, user data types, etc. Two non-**EQ** arrays may be **EQUALALL** if their respective componants are **EQUALALL**.

9.3 Logical Predicates

(AND X_1 X_2 ... X_N**)** [NLambda NoSpread Function]

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument evaluates to **NIL**, **AND** immediately returns **NIL** (without evaluating the remaining arguments). If all of the arguments evaluate to non-**NIL**, the value of the last argument is returned. **(AND) => T**.

(OR X_1 X_2 ... X_N**)** [NLambda NoSpread Function]

Takes an indefinite number of arguments (including zero), that are evaluated in order. If any argument is non-**NIL**, the value of that argument is returned by **OR** (without evaluating the remaining arguments). If all of the arguments evaluate to **NIL**, **NIL** is returned. **(OR) => NIL**.

AND and **OR** can be used as simple logical connectives, but note that they may not evaluate all of their arguments. This makes a difference if the evaluation of some of the arguments causes side-effects. Another result of this implementation of **AND** and **OR** is that they can be used as simple conditional statements. For example: **(AND (LISTP** *X***) (CDR** *X***))** returns the value of **(CDR** *X***)** if *X* is a list cell, otherwise it returns **NIL** without evaluating **(CDR** *X***)**. In general, this use of **AND** and **OR** should be avoided in favor of more explicit conditional statements in order to make programs more readable.

9.4 The COND Conditional Function

(COND $CLAUSE_1$ $CLAUSE_2$ ... $CLAUSE_K$**)** [NLambda NoSpread Function]

The conditional function of Interlisp, **COND**, takes an indefinite number of arguments, called clauses. Each $CLAUSE_i$ is a list of the form (P_i C_{i1} ... C_{iN}), where P_i is the predicate, and C_{i1} ... C_{iN} are the consequents. The operation of **COND** can be paraphrased as:

IF P_1 THEN C_{11} ... C_{1N} ELSEIF P_2 THEN C_{21} ... C_{2N} ELSEIF P_3 ...

The clauses are considered in sequence as follows: the predicate P_1 of the clause $CLAUSE_i$ is evaluated. If the value of P_1 is "true" (non-**NIL**), the consequents C_{i1} ... C_{iN} are evaluated in order, and the value of the **COND** is the value of C_{iN}, the last expression in the clause. If P_1 is "false" (**EQ** to **NIL**), then the remainder of $CLAUSE_i$ is ignored, and the next clause, $CLAUSE_{i+1}$, is considered. If no P_i is true for *any* clause, the value of the **COND** is **NIL**.

Note: If a clause has no consequents, and has the form (P_i), then if P_i evaluates to non-**NIL**, it is returned as the value of the **COND**. It is only evaluated once.

Example:

```
←(DEFINEQ (DOUBLE (X)
     (COND ((NUMBERP X) (PLUS X X))
```

```
        ((STRINGP X) (CONCAT X X))
        ((ATOM X) (PACK* X X))
        (T (PRINT "unknown") X)
        ((HORRIBLE-ERROR))]
(DOUBLE)
←(DOUBLE 5)
10
←(DOUBLE "FOO")
"FOOFOO"
←(DOUBLE 'BAR)
BARBAR
←(DOUBLE '(A B C))
"unknown"
(A B C)
```

A few points about this example: Notice that **5** is both a number and an atom, but it is "caught" by the **NUMBERP** clause before the **ATOM** clause. Also notice the predicate T, which is always true. This is the normal way to indicate a **COND** clause which will always be executed (if none of the preceeding clauses are true). **(HORRIBLE-ERROR)** will never be executed.

9.5 The IF Statement

The **IF** statement provides a way of way of specifying conditional expressions that is much easier and readable than using the **COND** function directly (page 9.4). CLISP translates expressions employing **IF**, **THEN**, **ELSEIF**, or **ELSE** (or their lowercase versions) into equivalent **COND** expressions. In general, statements of the form:

(if *AAA* **then** *BBB* **elseif** *CCC* **then** *DDD* **else** *EEE***)**

are translated to:

(COND (*AAA BBB***)**
   (*CCC DDD***)**
   (T *EEE***))**

The segment between **IF** or **ELSEIF** and the next **THEN** corresponds to the predicate of a **COND** clause, and the segment between **THEN** and the next **ELSE** or **ELSEIF** as the consequent(s). **ELSE** is the same as **ELSEIF T THEN**. These words are spelling corrected using the spelling list **CLISPIFWORDSPLST**. Lower case versions (**if, then, elseif, else**) may also be used.

If there is nothing following a **THEN**, or **THEN** is omitted entirely, then the resulting **COND** clause has a predicate but no consequent. For example, **(if X then elseif ...)** and **(if X elseif ...)**

both translate to **(COND (X) ...)**, which means that if **X** is not **NIL**, it is returned as the value of the **COND**.

Note that only one expression is allowed as the predicate, but multiple expressions are allowed as the consequents after **THEN** or **ELSE**. Multiple consequent expressions are implicitely wrapped in a **PROGN**, and the value of the last one is returned as the value of the consequent. For example:

(if X then (PRINT "FOO") (PRINT "BAR") elseif Y then (PRINT "BAZ"))

CLISP considers **IF**, **THEN**, **ELSE**, and **ELSEIF** to have lower precedence than all infix and prefix operators, as well as Interlisp forms, so it is sometimes possible to omit parentheses around predicate or consequent forms. For example, **(if FOO X Y then ...)** is equivalent to **(if (FOO X Y) then ...)**, and **(if X then FOO X Y else ...)** as equivalent to **(if X then (FOO X Y) else ...)**. Essentially, CLISP determines whether the segment between **THEN** and the next **ELSE** or **ELSEIF** corresponds to one form or several and acts accordingly, occasionally interacting with the user to resolve ambiguous cases. Note that if **FOO** is bound as a variable, **(if FOO then ...)** is translated as **(COND (FOO ...))**, so if a call to the *function* **FOO** is desired, use **(if (FOO) then ...)**.

9.6 Selection Functions

(SELECTQ *X* $CLAUSE_1$ $CLAUSE_2$ ... $CLAUSE_K$ *DEFAULT*) [NLambda NoSpread Function]

Selects a form or sequence of forms based on the value of *X*. Each clause $CLAUSE_i$ is a list of the form (S_i C_{i1} ... C_{iN}) where S_i is the selection key. The operation of **SELECTQ** can be paraphrased as:

IF $X = S_1$ THEN C_{11} ... C_{1N} ELSEIF $X = S_2$ THEN ... ELSE *DEFAULT*.

If S_i is an atom, the value of *X* is tested to see if it is **EQ** to S_i (which is not evaluated). If so, the expressions C_{i1} ... C_{iN} are evaluated in sequence, and the value of the **SELECTQ** is the value of the last expression evaluated, i.e., C_{iN}.

If S_i is a list, the value of *X* is compared with each element (not evaluated) of S_i, and if *X* is **EQ** to any one of them, then C_{i1} ... C_{iN} are evaluated as above.

If $CLAUSE_i$ is not selected in one of the two ways described, $CLAUSE_{i+1}$ is tested, etc., until all the clauses have been tested. If none is selected, *DEFAULT* is evaluated, and its value is returned as the value of the **SELECTQ**. *DEFAULT* must be present.

An example of the form of a **SELECTQ** is:

```
[SELECTQ MONTH
  (FEBRUARY (if (LEAPYEARP) then 29 else 28))
  ((APRIL JUNE SEPTEMBER NOVEMBER) 30)
  31]
```

If the value of **MONTH** is the litatom **FEBRUARY**, the **SELECTQ** returns 28 or 29 (depending on **(LEAPYEARP)**); otherwise if **MONTH** is **APRIL, JUNE, SEPTEMBER**, or **NOVEMBER**, the **SELECTQ** returns 30; otherwise it returns 31.

SELECTQ compiles open, and is therefore very fast; however, it will not work if the value of *X* is a list, a large integer, or floating point number, since **SELECTQ** uses **EQ** for all comparisons.

Note: **SELCHARQ** (page 2.15) is a version of **SELECTQ** that recognizes **CHARCODE** litatoms.

(SELECTC *X CLAUSE*$_1$ *CLAUSE*$_2$ ... *CLAUSE*$_K$ *DEFAULT***)** [NLambda NoSpread Function]

"**SELECTQ**-on-Constant." Similar to **SELECTQ** except that the selection keys are evaluated, and the result used as a **SELECTQ**-style selection key.

SELECTC is compiled as a **SELECTQ**, with the selection keys evaluated at compile-time. Therefore, the selection keys act like compile-time constants (see page 18.7). For example:

```
[SELECTC NUM
  ((for X from 1 to 9 collect (TIMES X X)) "SQUARE" )
  "HIP"]
```

compiles as:

```
[SELECTQ NUM
  ((1 4 9 16 25 36 49 64 81) "SQUARE" )
  "HIP"]
```

9.7 PROG and Associated Control Functions

(PROG1 *X*$_1$ *X*$_2$ ... *X*$_N$**)** [NLambda NoSpread Function]

Evaluates its arguments in order, and returns the value of its first argument *X*$_1$. For example, **(PROG1 X (SETQ X Y))** sets **X** to **Y**, and returns **X**'s original value.

(PROG2 *X*$_1$ *X*$_2$ ... *X*$_N$**)** [NoSpread Function]

Similar to **PROG1**. Evaluates its arguments in order, and returns the value of its second argument *X*$_2$.

(PROGN *X_1 X_2 ... X_N***)** [NLambda NoSpread Function]

PROGN evaluates each of its arguments in order, and returns the value of its last argument. **PROGN** is used to specify more than one computation where the syntax allows only one, e.g., **(SELECTQ ... (PROGN ...))** allows evaluation of several expressions as the default condition for a **SELECTQ**.

(PROG *VARLST E_1 E_2 ... E_N***)** [NLambda NoSpread Function]

This function allows the user to write an ALGOL-like program containing Interlisp expressions (forms) to be executed. The first argument, *VARLST*, is a list of local variables (must be **NIL** if no variables are used). Each atom in *VARLST* is treated as the name of a local variable and bound to **NIL**. *VARLST* can also contain lists of the form (*LITATOM FORM*). In this case, *LITATOM* is the name of the variable and is bound to the value of *FORM*. The evaluation takes place before any of the bindings are performed, e.g., **(PROG ((X Y) (Y X)) ...)** will bind local variable **X** to the value of **Y** (evaluated *outside* the **PROG**) and local variable **Y** to the value of **X** (outside the **PROG**). An attempt to use anything other than a litatom as a **PROG** variable will cause an error, **ARG NOT LITATOM**. An attempt to use **NIL** or **T** as a **PROG** variable will cause an error, **ATTEMPT TO BIND NIL OR T**.

The rest of the **PROG** is a sequence of non-atomic statements (forms) and litatoms (labels). The forms are evaluated sequentially; the labels serve only as markers. The two special functions **GO** and **RETURN** alter this flow of control as described below. The value of the **PROG** is usually specified by the function **RETURN**. If no **RETURN** is executed before the **PROG** "falls off the end," the value of the **PROG** is **NIL**.

(GO *U***)** [NLambda NoSpread Function]

GO is used to cause a transfer in a **PROG**. **(GO L)** will cause the **PROG** to evaluate forms starting at the label **L** (**GO** does not evaluate its argument). A **GO** can be used at any level in a **PROG**. If the label is not found, **GO** will search higher progs *within the same function,* e.g., **(PROG ... A ... (PROG ... (GO A)))**. If the label is not found in the function in which the **PROG** appears, an error is generated, **UNDEFINED OR ILLEGAL GO**.

(RETURN *X***)** [Function]

A **RETURN** is the normal exit for a **PROG**. Its argument is evaluated and is immediately returned the value of the **PROG** in which it appears.

Note: If a **GO** or **RETURN** is executed in an interpreted function which is not a **PROG**, the **GO** or **RETURN** will be executed in the last interpreted **PROG** entered if any, otherwise cause an error.

GO or **RETURN** inside of a compiled function that is not a **PROG** is not allowed, and will cause an error at compile time.

As a corollary, **GO** or **RETURN** in a functional argument, e.g., to **SORT**, will not work compiled. Also, since **NLSETQ**'s and **ERSETQ**'s compile as *separate* functions, a **GO** or **RETURN** *cannot* be used inside of a compiled **NLSETQ** or **ERSETQ** if the corresponding **PROG** is outside, i.e., above, the **NLSETQ** or **ERSETQ**.

(LET *VARLST* E_1 E_2 ... E_N**)** [Macro]

LET is essentially a **PROG** that can't contain **GO**'s or **RETURN**'s, and whose last form is the returned value.

(LET* *VARLST* E_1 E_2 ... E_N**)** [Macro]

(PROG* *VARLST* E_1 E_2 ... E_N**)** [Macro]

LET* and **PROG*** differ from **LET** and **PROG** only in that the binding of the bound variables is done "sequentially." Thus

```
(LET* ((A (LIST 5))
   (B (LIST A A)))
  (EQ A (CADR B)))
```

would evaluate to **T**; whereas the same form with **LET** might even find A an unbound variable when evaluating **(LIST A A)**.

9.8 The Iterative Statement

The iterative statement (i.s.) in its various forms permits the user to specify complicated iterative statements in a straightforward and visible manner. Rather than the user having to perform the mental transformations to an equivalent Interlisp form using **PROG, MAPC, MAPCAR,** etc., the system does it for him. The goal was to provide a robust and tolerant facility which could "make sense" out of a wide class of iterative statements. Accordingly, the user should not feel obliged to read and understand in detail the description of each operator given below in order to use iterative statements.

An iterative statement is a form consisting of a number of special words (known as i.s. operators or i.s.oprs), followed by operands. Many i.s.oprs (**FOR**, **DO**, **WHILE**, etc.) are similar to iterative statements in other programming languages; other i.s.oprs (**COLLECT**, **JOIN**, **IN**, etc.) specify useful operations in a Lisp environment. Lower case versions of i.s.oprs (**do**, **collect**, etc.)

can also be used. Here are some examples of iterative statements:

```
←(for X from 1 to 5 do (PRINT 'FOO))
FOO
FOO
FOO
FOO
FOO
NIL
←(for X from 2 to 10 by 2 collect (TIMES X X))
(4 16 36 64 100)
←(for X in '(A B 1 C 6.5 NIL (45)) count (NUMBERP X))
2
```

Iterative statements are implemented through CLISP, which translates the form into the appropriate **PROG**, **MAPCAR**, etc. Iterative statement forms are translated using all CLISP declarations in effect (standard/fast/undoable/ etc.); see page 21.12. Misspelled i.s.oprs are recognized and corrected using the spelling list **CLISPFORWORDSPLST**. The order of appearance of operators is never important; CLISP scans the entire statement before it begins to construct the equivalent Interlisp form. New i.s.oprs can be defined as described on page 9.20.

If the user defines a function by the same name as an i.s.opr (**WHILE**, **TO**, etc.), the i.s.opr will no longer have the CLISP interpretation when it appears as **CAR** of a form, although it will continue to be treated as an i.s.opr if it appears in the interior of an iterative statement. To alert the user, a warning message is printed, e.g., **(WHILE DEFINED, THEREFORE DISABLED IN CLISP)**.

9.8.1 I.s.types

The following i.s.oprs are examples of a certain kind of iterative statement operator called an i.s.type. The i.s.type specifies what is to be done at each iteration. Its operand is called the "body" of the iterative statement. Each iterative statement must have one and only one i.s.type.

DO *FORM* [I.S. Operator]

Specifies what is to be done at each iteration. **DO** with no other operator specifies an infinite loop. If some explicit or implicit terminating condition is specified, the value of the i.s. is **NIL**. Translates to **MAPC** or **MAP** whenever possible.

COLLECT *FORM* [I.S. Operator]

Specifies that the value of *FORM* at each iteration is to be collected in a list, which is returned as the value of the i.s. when it terminates. Translates to **MAPCAR**, **MAPLIST** or **SUBSET** whenever possible.

When **COLLECT** translates to a **PROG** (e.g., if **UNTIL**, **WHILE**, etc. appear in the i.s.), the translation employs an open **TCONC** using two pointers similar to that used by the compiler for compiling **MAPCAR**. To disable this translation, perform **(CLDISABLE 'FCOLLECT)** (see page 21.26).

JOIN *FORM* [I.S. Operator]

Similar to **COLLECT**, except that the values of *FORM* at each iteration are **NCONC**ed. Translates to **MAPCONC** or **MAPCON** whenever possible. /**NCONC**, /**MAPCONC**, and /**MAPCON** are used when the CLISP declaration **UNDOABLE** is in effect.

SUM *FORM* [I.S. Operator]

Specifies that the values of *FORM* at each iteration be added together and returned as the value of the i.s., e.g., **(for I from 1 to 5 sum (TIMES I I))** returns 1 + 4 + 9 + 16 + 25 = 55. **IPLUS**, **FPLUS**, or **PLUS** will be used in the translation depending on the CLISP declarations in effect.

COUNT *FORM* [I.S. Operator]

Counts the number of times that *FORM* is true, and returns that count as its value.

ALWAYS *FORM* [I.S. Operator]

Returns **T** if the value of *FORM* is non-**NIL** for all iterations. (Note: returns **NIL** as soon as the value of *FORM* is **NIL**).

NEVER *FORM* [I.S. Operator]

Similar to **ALWAYS**, except returns **T** if the value of *FORM* is *never* true. (Note: returns **NIL** as soon as the value of *FORM* is non-**NIL**).

The following i.s.types explicitly refer to the iteration variable (i.v.) of the iterative statement, which is a variable set at each iteration. This is explained below under **FOR**.

THEREIS *FORM* [I.S. Operator]

Returns the first value of the i.v. for which *FORM* is non-**NIL**, e.g., **(for X in Y thereis (NUMBERP X))** returns the first number in **Y**. (Note: returns the value of the i.v. as soon as the value of *FORM* is non-**NIL**).

LARGEST *FORM* [I.S. Operator]

SMALLEST *FORM* [I.S. Operator]

Returns the value of the i.v. that provides the largest/smallest value of *FORM*. **$$EXTREME** is always bound to the current greatest/smallest value, **$$VAL** to the value of the i.v. from which it came.

9.8.2 Iteration Variable I.s.oprs

FOR *VAR* [I.S. Operator]

Specifies the iteration variable (i.v.) which is used in conjunction with **IN, ON, FROM, TO,** and **BY**. The variable is rebound within the i.s., so the value of the variable outside the i.s. is not effected. Example:

```
←(SETQ X 55)
55
←(for X from 1 to 5 collect (TIMES X X))
(1 4 9 16 25)
←X
55
```

FOR *VARS* [I.S. Operator]

VARS a list of variables, e.g., **(for (X Y Z) in ...)**. The first variable is the i.v., the rest are dummy variables. See **BIND** below.

FOR OLD *VAR* [I.S. Operator]

Similar to **FOR**, except that *VAR* is *not* rebound within the i.s., so the value of the i.v. outside of the i.s. is changed. Example:

```
←(SETQ X 55)
55
←(for old X from 1 to 5 collect (TIMES X X))
(1 4 9 16 25)
←X
6
```

BIND *VAR* [I.S. Operator]

BIND *VARS* [I.S. Operator]

Used to specify dummy variables, which are bound locally within the i.s.

Note: **FOR, FOR OLD,** and **BIND** variables can be initialized by using the form *VAR←FORM*:

(for old (X←*FORM*) bind (Y←*FORM*) ...)

IN *FORM* [I.S. Operator]

Specifies that the i.s. is to iterate down a list with the i.v. being reset to the corresponding element at each iteration. For example, **(for X in Y do ...)** corresponds to **(MAPC Y (FUNCTION (LAMBDA (X) ...)))**. If no i.v. has been specified, a dummy is supplied, e.g., **(in Y collect CADR)** is equivalent to **(MAPCAR Y (FUNCTION CADR))**.

ON *FORM* [I.S. Operator]

Same as **IN** except that the i.v. is reset to the corresponding *tail* at each iteration. Thus **IN** corresponds to **MAPC**, **MAPCAR**, and **MAPCONC**, while **ON** corresponds to **MAP**, **MAPLIST**, and **MAPCON**.

Note: for both **IN** and **ON**, *FORM* is evaluated before the main part of the i.s. is entered, i.e. *outside* of the scope of any of the bound variables of the i.s. For example, **(for X bind (Y←'(1 2 3)) in Y ...)** will map down the list which is the value of **Y** evaluated *outside* of the i.s., *not* **(1 2 3)**.

IN OLD *VAR* [I.S. Operator]

Specifies that the i.s. is to iterate down *VAR*, with *VAR* itself being reset to the corresponding tail at each iteration, e.g., after **(for X in old L do ... until ...)** finishes, **L** will be some tail of its original value.

IN OLD (*VAR*←*FORM***)** [I.S. Operator]

Same as **IN OLD** *VAR*, except *VAR* is first set to value of *FORM*.

ON OLD *VAR* [I.S. Operator]

Same as **IN OLD** *VAR* except the i.v. is reset to the current value of *VAR* at each iteration, instead of to **(CAR** *VAR***)**.

ON OLD (*VAR*←*FORM***)** [I.S. Operator]

Same as **ON OLD** *VAR*, except *VAR* is first set to value of *FORM*.

INSIDE *FORM* [I.S. Operator]

Similar to **IN**, except treats first non-list, non-**NIL** tail as the last element of the iteration, e.g., **INSIDE '(A B C D . E)** iterates five times with the i.v. set to **E** on the last iteration. **INSIDE 'A** is equivalent to **INSIDE '(A)**, which will iterate once.

FROM *FORM* [I.S. Operator]

Used to specify an initial value for a numerical i.v. The i.v. is automatically incremented by 1 after each iteration (unless **BY** is specified). If no i.v. has been specified, a dummy i.v. is supplied and initialized, e.g., **(from 2 to 5 collect SQRT)** returns **(1.414 1.732 2.0 2.236)**.

TO *FORM* [I.S. Operator]

Used to specify the final value for a numerical i.v. If **FROM** is not specified, the i.v. is initialized to 1. If no i.v. has been specified, a dummy i.v. is supplied and initialized. If **BY** is not specified, the i.v. is automatically incremented by 1 after each iteration. When the i.v. is definitely being *incremented*, i.e., either **BY** is not specified, or its operand is a positive number, the i.s. terminates when the i.v. exceeds the value of *FORM*. Similarly, when the i.v. is definitely being decremented the i.s. terminates when the i.v. becomes *less* than the value of *FORM* (see description of **BY**).

Note: *FORM* is evaluated only once, when the i.s. is first entered, and its value bound to a temporary variable against which the i.v. is checked each interation. If the user wishes to specify an i.s. in which the value of the boundary condition is recomputed each iteration, he should use **WHILE** or **UNTIL** instead of **TO**.

Note: When both the operands to **TO** and **FROM** are numbers, and **TO**'s operand is less than **FROM**'s operand, the i.v. is decremented by 1 after each iteration. In this case, the i.s. terminates when the i.v. becomes *less* than the value of *FORM*. For example, **(from 10 to 1 do PRINT)** prints the numbers from 10 down to 1.

BY *FORM* (with **IN/ON**) [I.S. Operator]

If **IN** or **ON** have been specified, the value of *FORM* determines the *tail* for the next iteration, which in turn determines the value for the i.v. as described earlier, i.e., the new i.v. is **CAR** of the tail for **IN**, the tail itself for **ON**. In conjunction with **IN**, the user can refer to the current tail within *FORM* by using the i.v. or the operand for **IN/ON**, e.g., **(for Z in L by (CDDR *Z*) ...)** or **(for Z in L by (CDDR *L*) ...)**. At translation time, the name of the internal variable which holds the value of the current tail is substituted for the i.v. throughout *FORM*. For example, **(for X in Y by (CDR (MEMB 'FOO (CDR X))) collect X)** specifies that after each iteration, **CDR** of the current tail is to be searched for the atom **FOO**, and (**CDR** of) this latter tail to be used for the next iteration.

BY *FORM* (without **IN/ON**) [I.S. Operator]

If **IN** or **ON** have not been used, **BY** specifies how the i.v. itself is reset at each iteration. If **FROM** or **TO** have been specified, the i.v. is known to be numerical, so the new i.v. is computed by adding the value of *FORM* (which is reevaluated each iteration) to the current value of the i.v., e.g., **(for N from 1 to 10 by 2 collect N)** makes a list of the first five odd numbers.

If *FORM* is a positive number (*FORM* itself, not its value, which in general CLISP would have no way of knowing in advance), the i.s. terminates when the value of the i.v. *exceeds* the value of **TO**'s operand. If *FORM* is a negative number, the i.s. terminates when the value of the i.v. becomes *less* than **TO**'s operand, e.g., **(for I from N to M by -2 until (LESSP I M) ...)**. Otherwise, the terminating condition for each iteration depends on the value of *FORM* for that iteration: if *FORM*<0, the test is whether the i.v. is less than **TO**'s operand, if *FORM*>0 the test is whether the i.v. exceeds **TO**'s operand, otherwise if *FORM* = 0, the i.s. terminates unconditionally.

If **FROM** or **TO** have not been specified and *FORM* is not a number, the i.v. is simply reset to the value of *FORM* after each iteration, e.g., **(for I from N by M ...)** is equivalent to **(for I←N by (PLUS I M) ...)**.

AS *VAR* [I.S. Operator]

Used to specify an iterative statement involving more than one iterative variable, e.g., **(for X in Y as U in V do ...)** corresponds to **MAP2C** (page 10.16). The i.s. terminates when any of the terminating conditions are met, e.g., **(for X in Y as I from 1 to 10 collect X)** makes a list of the first ten elements of **Y**, or however many elements there are on **Y** if less than 10.

The operand to **AS**, *VAR*, specifies the new i.v. For the remainder of the i.s., or until another **AS** is encountered, all operators refer to the new i.v. For example, **(for I from 1 to N1 as J from 1 to N2 by 2 as K from N3 to 1 by -1 ...)** terminates when **I** exceeds **N1**, or **J** exceeds **N2**, or **K** becomes less than 1. After each iteration, **I** is incremented by 1, **J** by 2, and **K** by -1.

OUTOF *FORM* [I.S. Operator]

For use with generators (page 11.16). On each iteration, the i.v. is set to successive values returned by the generator. The i.s. terminates when the generator runs out.

9.8.3 Condition I.s.oprs

WHEN *FORM* [I.S. Operator]

Provides a way of excepting certain iterations. For example, **(for X in Y collect X when (NUMBERP X))** collects only the elements of **Y** that are numbers.

UNLESS *FORM* [I.S. Operator]

Same as **WHEN** except for the difference in sign, i.e., **WHEN Z** is the same as **UNLESS (NOT Z)**.

WHILE *FORM* [I.S. Operator]

Provides a way of terminating the i.s. **WHILE** *FORM* evaluates *FORM before* each iteration, and if the value is **NIL**, exits.

UNTIL *FORM* [I.S. Operator]

Same as **WHILE** except for difference in sign, i.e., **WHILE X** is equivalent to **UNTIL (NOT X)**.

UNTIL *N* **(***N* **a number)** [I.S. Operator]

Equivalent to **UNTIL** *I.V.* **>** *N.*

REPEATWHILE *FORM* [I.S. Operator]

Same as **WHILE** except the test is performed after the evalution of the body, but before the i.v. is reset for the next iteration.

REPEATUNTIL *FORM* [I.S. Operator]

Same as **UNTIL**, except the test is performed after the evaluation of the body.

REPEATUNTIL *N* **(***N* **a number)** [I.S. Operator]

Equivalent to **REPEATUNTIL** *I.V.* **>** *N.*

9.8.4 Other I.s.oprs

FIRST *FORM* [I.S. Operator]

FORM is evaluated once before the first iteration, e.g., **(for X Y Z in L first (FOO Y Z) ...)**, and **FOO** could be used to initialize **Y** and **Z**.

FINALLY *FORM* [I.S. Operator]

FORM is evaluated after the i.s. terminates. For example, **(for X in L bind Y←0 do (if (ATOM X) then (SETQ Y (PLUS Y 1))) finally (RETURN Y))** will return the number of atoms in **L**.

EACHTIME *FORM* [I.S. Operator]

FORM is evaluated at the beginning of each iteration before, and regardless of, any testing. For example, consider,

```
(for I from 1 to N
  do (... (FOO I) ...)
```

unless (... (FOO I) ...)
until (... (FOO I) ...))

The user might want to set a temporary variable to the value of **(FOO I)** in order to avoid computing it three times each iteration. However, without knowing the translation, he would not know whether to put the assignment in the operand to **DO**, **UNLESS**, or **UNTIL**, i.e., which one would be executed first. He can avoid this problem by simply writing **EACHTIME (SETQ J (FOO I))**.

DECLARE: *DECL* [I.S. Operator]

Inserts the form **(DECLARE** *DECL***)** immediately following the **PROG** variable list in the translation, or, in the case that the translation is a mapping function rather than a **PROG**, immediately following the argument list of the lambda expression in the translation. This can be used to declare variables bound in the iterative statement to be compiled as local or special variables (see page 18.5). For example **(for X in Y declare: (LOCALVARS X) ...)**. Several **DECLARE:**s can apppear in the same i.s.; the declarations are inserted in the order they appear.

DECLARE *DECL* [I.S. Operator]

Same as **DECLARE:**.

Note that since **DECLARE** is also the name of a function, **DECLARE** cannot be used as an i.s. operator when it appears as **CAR** of a form, i.e. as the first i.s. operator in an iterative statement. However, **declare** (lower-case version) *can* be the first i.s. operator.

ORIGINAL *I.S.OPR OPERAND* [I.S. Operator]

I.S.OPR will be translated using its original, built-in interpretation, independent of any user defined i.s. operators. See page 9.20.

There are also a number of i.s.oprs that make it easier to create iterative statements that use the clock, looping for a given period of time. See timers, page 12.16.

9.8.5 Miscellaneous Hints on I.S.Oprs

- Lowercase versions of all i.s. operators are equivalent to the uppercase, e.g., **(for X in Y ...)** is equivalent to **(FOR X IN Y ...)**.
- Each i.s. operator is of lower precedence than all Interlisp forms, so parentheses around the operands can be omitted, and will be supplied where necessary, e.g., **BIND (X Y Z)** can be written **BIND**

X Y Z, OLD (X←*FORM*) as **OLD X←*FORM*, WHEN (NUMBERP X)** as **WHEN NUMBERP X,** etc.

- **RETURN** or **GO** may be used in any operand. (In this case, the translation of the iterative statement will always be in the form of a **PROG**, never a mapping function.) **RETURN** means return from the i.s. (with the indicated value), *not* from the function in which the i.s appears. **GO** refers to a label elsewhere in the function in which the i.s. appears, except for the labels **$$LP**, **$$ITERATE**, and **$$OUT** which are reserved, as described below.

- In the case of **FIRST, FINALLY, EACHTIME, DECLARE:** or one of the i.s.types, e.g., **DO, COLLECT, SUM,** etc., the operand can consist of more than one form, e.g., **COLLECT (PRINT (CAR X)) (CDR X),** in which case a **PROGN** is supplied.

- Each operand can be the name of a function, in which case it is applied to the (last) i.v., e.g., **(for X in Y do PRINT when NUMBERP)** is the same as **(for X in Y do (PRINT X) when (NUMBERP X)).** Note that the i.v. need not be explicitly specified, e.g., **(in Y do PRINT when NUMBERP)** will work.

  For i.s.types, e.g., **DO, COLLECT, JOIN,** the function is always applied to the first i.v. in the i.s., whether explicity named or not. For example, **(in Y as I from 1 to 10 do PRINT)** prints elements on **Y,** not integers between 1 and 10.

  Note that this feature does not make much sense for **FOR, OLD, BIND, IN,** or **ON,** since they "operate" before the loop starts, when the i.v. may not even be bound.

  In the case of **BY** in conjunction with **IN,** the function is applied to the current *tail* e.g., **(for X in Y by CDDR ...)** is the same as **(for X in Y by (CDDR X) ...).**

- While the exact form of the translation of an iterative statement depends on which operators are present, a **PROG** will always be used whenever the i.s. specifies dummy variables, i.e., if a **BIND** operator appears, or there is more than one variable specified by a **FOR** operator, or a **GO, RETURN,** or a reference to the variable **$$VAL** appears in any of the operands. When a **PROG** is used, the form of the translation is:

```
(PROG VARIABLES
   {initialize}
$$LP {eachtime}
   {test}
   {body}
$$ITERATE
   {aftertest}
   {update}
   (GO $$LP)
$$OUT {finalize}
   (RETURN $$VAL))
```

where **{test}** corresponds to that portion of the loop that tests for termination and also for those iterations for which **{body}** is not going to be executed, (as indicated by a **WHEN** or **UNLESS**); **{body}** corresponds to the operand of the i.s.type, e.g., **DO**, **COLLECT**, etc.; **{aftertest}** corresponds to those tests for termination specified by **REPEATWHILE** or **REPEATUNTIL**; and **{update}** corresponds to that part that resets the tail, increments the counter, etc. in preparation for the next iteration. **{initialize}**, **{finalize}**, and **{eachtime}** correspond to the operands of **FIRST**, **FINALLY**, and **EACHTIME**, if any.

Note that since **{body}** always appears at the top level of the **PROG**, the user can insert labels in **{body}**, and **GO** to them from within **{body}** or from other i.s. operands, e.g., **(for X in Y first (GO A) do (FOO) A (FIE))**. However, since **{body}** is dwimified as a list of forms, the label(s) should be added to the dummy variables for the iterative statement in order to prevent their being dwimified and possibly "corrected", e.g., **(for X in Y bind A first (GO A) do (FOO) A (FIE))**. The user can also **GO** to **$$LP**, **$$ITERATE**, or **$$OUT**, or explicitly set **$$VAL**.

9.8.6 Errors in Iterative Statements

An error will be generated and an appropriate diagnostic printed if any of the following conditions hold:

(1) Operator with null operand, i.e., two adjacent operators, as in **(for X in Y until do ...)**

(2) Operand consisting of more than one form (except as operand to **FIRST**, **FINALLY**, or one of the i.s.types), e.g., **(for X in Y (PRINT X) collect ...)**.

(3) **IN**, **ON**, **FROM**, **TO**, or **BY** appear twice in same i.s.

(4) Both **IN** and **ON** used on same i.v.

(5) **FROM** or **TO** used with **IN** or **ON** on same i.v.

(6) More than one i.s.type, e.g., a **DO** and a **SUM**.

In 3, 4, or 5, an error is not generated if an intervening **AS** occurs.

If an error occurs, the i.s. is left unchanged.

If no **DO**, **COLLECT**, **JOIN** or any of the other i.s.types are specified, CLISP will first attempt to find an operand consisting of more than one form, e.g., **(for X in Y (PRINT X) when ATOM X ...)**, and in this case will insert a **DO** after the first form. (In this case, condition 2 is not considered to be met, and an error is not generated.) If CLISP cannot find such an operand, and no **WHILE** or **UNTIL** appears in the i.s., a warning message is printed: **NO DO, COLLECT, OR JOIN:** followed by the i.s.

Similarly, if no terminating condition is detected, i.e., no **IN**, **ON**, **WHILE**, **UNTIL**, **TO**, or a **RETURN** or **GO**, a warning message is printed: **POSSIBLE NON-TERMINATING ITERATIVE STATEMENT:** followed by the iterative statement. However, since the user may be planning to terminate the i.s. via an error, control-E, or a **RETFROM** from a lower function, the i.s. is still translated. Note: The error message is not printed if the value of **CLISPI.S.GAG** is **T** (initially **NIL**).

9.8.7 Defining New Iterative Statement Operators

The following function is available for defining new or redefining existing iterative statement operators:

(I.S.OPR *NAME FORM OTHERS EVALFLG***)** [Function]

NAME is the name of the new i.s.opr. If *FORM* is a list, *NAME* will be a new *i.s.type* (see page 9.10), and *FORM* its body.

OTHERS is an (optional) list of additional i.s. operators and operands which will be added to the i.s. at the place where *NAME* appears. If *FORM* is **NIL**, *NAME* is a new i.s.opr defined entirely by *OTHERS*.

In both *FORM* and *OTHERS*, the atom **$$VAL** can be used to reference the value to be returned by the i.s., **I.V.** to reference the current i.v., and **BODY** to reference *NAME*'s operand. In other words, the current i.v. will be substituted for all instances of **I.V.** and *NAME*'s operand will be substituted for all instances of **BODY** throughout *FORM* and *OTHERS*.

If *EVALFLG* is **T**, *FORM* and *OTHERS* are evaluated at translation time, and their values used as described above. A dummy variable for use in translation that does not clash with a dummy variable already used by some other i.s. operators can be obtained by calling **(GETDUMMYVAR)**. **(GETDUMMYVAR T)** will return a dummy variable and also insure that it is bound as a **PROG** variable in the translation.

If *NAME* was previously an i.s.opr and is being redefined, the message **(***NAME* **REDEFINED)** will be printed (unless **DFNFLG = T**), and all expressions using the i.s.opr *NAME* that have been translated will have their translations discarded.

The following are some examples of how **I.S.OPR** could be called to define some existing i.s.oprs, and create some new ones:

COLLECT

```
(I.S.OPR 'COLLECT
    '(SETQ $$VAL (NCONC1 $$VAL BODY)))
```

SUM

```
(I.S.OPR 'SUM
    '($$VAL←$$VAL + BODY)
    '(FIRST $$VAL←0))
```

Note: **$$VAL + BODY** is used instead of **(IPLUS $$VAL BODY)** so that the choice of function used in the translation, i.e., **IPLUS**, **FPLUS**, or **PLUS**, will be determined by the declarations then in effect.

NEVER

```
(I.S.OPR 'NEVER
    '(if BODY then $$VAL←NIL (GO $$OUT)))
```

Note: **(if BODY then (RETURN NIL))** would exit from the i.s. immediately and therefore not execute the operations specified via a **FINALLY** (if any).

THEREIS

```
(I.S.OPR 'THEREIS
    '(if BODY then $$VAL←I.V. (GO $$OUT)))
```

RCOLLECT To define **RCOLLECT**, a version of **COLLECT** which uses **CONS** instead of **NCONC1** and then reverses the list of values:

```
(I.S.OPR 'RCOLLECT
    '($$VAL←(CONS BODY $$VAL))
    '(FINALLY (RETURN (DREVERSE $$VAL)))]
```

TCOLLECT To define **TCOLLECT**, a version of **COLLECT** which uses **TCONC**:

```
(I.S.OPR 'TCOLLECT
    '(TCONC $$VAL BODY)
    '(FIRST $$VAL←(CONS) FINALLY (RETURN (CAR $$VAL)))]
```

PRODUCT

```
(I.S.OPR 'PRODUCT
    '($$VAL←$$VAL*BODY)
    '(FIRST $$VAL←1)]
```

UPTO To define **UPTO**, a version of **TO** whose operand is evaluated only once:

```
(I.S.OPR 'UPTO
    NIL
    '(BIND $$FOO←BODY TO $$FOO)]
```

TO To redefine **TO** so that instead of recomputing *FORM* each iteration, a variable is bound to the value of *FORM*, and then that variable is used:

```
(I.S.OPR 'TO
    NIL
    '(BIND $$END FIRST $$END←BODY ORIGINAL TO $$END)]
```

Note the use of **ORIGINAL** to redefine **TO** in terms of its original definition. **ORIGINAL** is intended for use in redefining built-in operators, since their definitions are not accessible, and hence not directly modifiable. Thus if the operator had been defined by the user via **I.S.OPR**, **ORIGINAL** would not obtain its original definition. In this case, one presumably would simply modify the i.s.opr definition.

I.S.OPR can also be used to define synonyms for already defined i.s. operators by calling **I.S.OPR** with *FORM* an atom, e.g., **(I.S.OPR 'WHERE 'WHEN)** makes **WHERE** be the same as **WHEN**. Similarly, following **(I.S.OPR 'ISTHERE 'THEREIS)**, one can write **(ISTHERE ATOM IN Y)**, and following **(I.S.OPR 'FIND 'FOR)** and **(I.S.OPR 'SUCHTHAT 'THEREIS)**, one can write **(find X in Y suchthat X member Z)**. In the current system, **WHERE** is synonymous with **WHEN**, **SUCHTHAT** and **ISTHERE** with **THEREIS**, **FIND** with **FOR**, and **THRU** with **TO**.

If *FORM* is the atom **MODIFIER**, then *NAME* is defined as an i.s.opr which can immediately follow another i.s. operator (i.e., an error will not be generated, as described previously). *NAME* will not terminate the scope of the previous operator, and will be stripped off when **DWIMIFY** is called on its operand. **OLD** is an example of a **MODIFIER** type of operator. The **MODIFIER** feature allows the user to define i.s. operators similar to **OLD**, for use in conjunction with some other user defined i.s.opr which will produce the appropriate translation.

The file package command **I.S.OPRS** (page 17.39) will dump the definition of i.s.oprs. **(I.S.OPRS PRODUCT UPTO)** as a file package command will print suitable expressions so that these iterative statement operators will be (re)defined when the file is loaded.

[This page intentionally left blank]

10. FUNCTION DEFINITION, MANIPULATION, AND EVALUATION

The Interlisp programming system is designed to help the user define and debug functions. Developing an applications program in Interlisp involves defining a number of functions in terms of the system primitives and other user-defined functions. Once defined, the user's functions may be referenced exactly like Interlisp primitive functions, so the programming process can be viewed as extending the Interlisp language to include the required functionality.

Functions are defined with a list expressions known as an "expr definition." An expr definition specifies if the function has a fixed or variable number of arguments, whether these arguments are evaluated or not, the function argument names, and a series of forms which define the behavior of the function. For example:

(LAMBDA (X Y) (PRINT X) (PRINT Y))

A function defined with this expr definition would have two evaluated arguments, **X** and **Y**, and it would execute **(PRINT X)** and **(PRINT Y)** when evaluated. Other types of expr definitions are described below.

A function is defined by putting an expr definition in the function definition cell of a litatom. There are a number of functions for accessing and setting function definition cells, but one usually defines a function with **DEFINEQ** (page 10.9). For example:

```
←(DEFINEQ (FOO (LAMBDA (X Y) (PRINT X) (PRINT Y))))
(FOO)
```

The expression above will define the function **FOO** to have the expr definition **(LAMBDA (X Y) (PRINT X) (PRINT Y))**. After being defined, this function may be evaluated just like any system function:

```
←(FOO 3 (IPLUS 3 4))
3
7
7
←
```

All function definition cells do not contain expr definitions. The compiler (page 18.1) translates expr definitions into compiled code objects, which execute much faster. Interlisp provides a number of "function type functions" which determine how a given function is defined, the number and names of function arguments, etc. See page 10.7.

Usually, functions are evaluated automatically when they appear within another function or when typed into Interlisp. However, sometimes it is useful to envoke the Interlisp interpreter explicitly to apply a given "functional argument" to some data. There are a number of functions which will apply a given function repeatedly. For example, **MAPCAR** will apply a function (or an expr definition) to all of the elements of a list, and return the values returned by the function:

←(MAPCAR '(1 2 3 4 5) '(LAMBDA (X) (ITIMES X X))
(1 4 9 16 25)

When using functional arguments, there are a number of problems which can arise, related with accessing free variables from within a function argument. Many times these problems can be solved using the function **FUNCTION** to create a **FUNARG** object (see page 10.18).

The macro facility provides another way of specifying the behavior of a function (see page 10.21). Macros are very useful when developing code which should run very quickly, which should be compiled differently than it is interpreted, or which should run differently in different implementations of Interlisp.

10.1 Function Types

Interlisp functions are defined using list expressions called "expr definitions." An expr definition is a list of the form (*LAMBDA-WORD ARG-LIST FORM*$_1$ ... *FORM*$_N$). *LAMBDA-WORD* determines whether the arguments to this function will be evaluated or not, *ARG-LIST* determines the number and names of arguments, and *FORM*$_1$ ... *FORM*$_N$ are a series of forms to be evaluated after the arguments are bound to the local variables in *ARG-LIST*.

If *LAMBDA-WORD* is the litatom **LAMBDA**, then the arguments to the function are evaluated. If *LAMBDA-WORD* is the litatom **NLAMBDA**, then the arguments to the function are not evaluated. Functions which evaluate or don't evaluate their arguments are therefore known as "lambda" or "nlambda" functions, respectively.

If *ARG-LIST* is **NIL** or a list of litatoms, this indicates a function with a fixed number of arguments. Each litatom is the name of an argument for the function defined by this expression. The process of binding these litatoms to the individual arguments is called "spreading" the arguments, and the function is called a "spread" function. If the argument list is any litatom other than **NIL**, this indicates a function with a variable number of arguments, known as a "nospread" function.

If *ARG-LIST* is anything other than a litatom or a list of litatoms, such as **(LAMBDA "FOO" ...)**, attempting to use this expr definition will generate an **ARG NOT LITATOM** error. In addition, if **NIL** or **T** is used as an argument name, the error **ATTEMPT TO BIND NIL OR T** is generated.

These two parameters (lambda/nlambda and spread/nospread) may be specified independently, so there are four main function types, known as lambda-spread, nlambda-spread, lambda-nospread, and nlambda-nospread functions. Each one has a different form, and is used for a different purpose. These four function types are described more fully below.

Note: For lambda-spread, lambda-nospread, or nlambda-spread functions, there is an upper limit to the number of arguments that a function can have, based on the number of arguments that can be stored on the stack on any one function call: Currently, the limit is 80 arguments. If a function is called with more than that many arguments, the error **TOO MANY ARGUMENTS** occurs. However, nlambda-nospread functions can be called with an arbitrary number of arguments, since the arguments are not individually saved on the stack (see page 10.6).

10.1.1 Lambda-Spread Functions

Lambda-spread functions take a fixed number of evaluated arguments. This is the most common function type. A lambda-spread expr definition has the form:

(LAMBDA (ARG_1 ... ARG_M**)** $FORM_1$ ... $FORM_N$**)**

The argument list **(**ARG_1 ... ARG_M**)** is a list of litatoms that gives the number and names of the formal arguments to the function. If the argument list is **()** or **NIL**, this indicates that the function takes no arguments. When a lambda-spread function is applied to some arguments, the arguments are evaluated, and bound to the local variables ARG_1 ... ARG_M. Then, $FORM_1$ ... $FORM_N$ are evaluated in order, and the value of the function is the value of $FORM_N$.

```
←(DEFINEQ (FOO (LAMBDA (X Y) (LIST X Y))))
(FOO)
←(FOO 99 (PLUS 3 4))
(99 7)
```

In the above example, the function **FOO** defined by **(LAMBDA (X Y) (LIST X Y))** is applied to the arguments **99** and **(PLUS 3 4)**, these arguments are evaluated (giving 99 and 7), the local variable **X** is bound to 99 and **Y** to 7, **(LIST X Y)** is evaluated, returning **(99 7)**, and this is returned as the value of the function.

A standard feature of the Interlisp system is that no error occurs if a spread function is called with too many or too few arguments. If a function is called with too many arguments, the extra arguments are evaluated but ignored. If a function is called with too few arguments, the unsupplied ones will be delivered as **NIL**. In fact, a spread function cannot distinguish between being given **NIL** as an argument, and not being given that argument, e.g., **(FOO)** and **(FOO NIL)** are *exactly* the same for spread functions. If it is necessary to distinguish between these two cases, use an nlambda function and explicitly evaluate the arguments with the **EVAL** function (page 10.12).

10.1.2 Nlambda-Spread Functions

Nlambda-spread functions take a fixed number of unevaluated arguments. An nlambda-spread expr definition has the form:

(NLAMBDA (ARG_1 ... ARG_M**)** $FORM_1$ ... $FORM_N$**)**

Nlambda-spread functions are evaluated similarly to lambda-spread functions, except that the arguments are not evaluated before being bound to the variables ARG_1 ... ARG_M.

```
←(DEFINEQ (FOO (NLAMBDA (X Y) (LIST X Y))) )
(FOO)
←(FOO 99 (PLUS 3 4))
(99 (PLUS 3 4))
```

In the above example, the function **FOO** defined by **(NLAMBDA (X Y) (LIST X Y))** is applied to the arguments **99** and **(PLUS 3 4)**, these arguments are bound unevaluated to **X** and **Y**, **(LIST X Y)** is evaluated, returning **(99 (PLUS 3 4))**, and this is returned as the value of the function.

Note: Functions can be defined so that all of their arguments are evaluated (lambda functions) or none are evaluated (nlambda functions). If it is desirable to write a function which only evaluates *some* of its arguments (e.g. **SETQ**), the function should be defined as an nlambda, with some arguments explicitly evaluated using the function **EVAL** (page 10.12). If this is done, the user should put the litatom **EVAL** on the property list of the function under the property **INFO**. This informs various system packages such as DWIM, CLISP, and Masterscope that this function in fact *does* evaluate its arguments, even though it is an nlambda.

Warning: A frequent problem that occurs when evaluating arguments to nlambda functions with **EVAL** (page 10.12) is that the form being evaluated may reference variables that are not accessable within the nlambda function. This is usually not a problem when interpreting code, but when the code is compiled, the values of "local" variables may not be accessable on the stack (see page 18.5). The system nlambda functions that evaluate their arguments (such as **SETQ**) are expanded in-line by the compiler, so this is not a problem. Using the macro facility (page 10.21) is recommended in cases where it is necessary to evaluate some arguments to an nlambda function.

10.1.3 Lambda-Nospread Functions

Lambda-nospread functions take a variable number of evaluated arguments. A lambda-nospread expr definition has the form:

(LAMBDA *VAR* $FORM_1$ ... $FORM_N$**)**

VAR may be any litatom, except **NIL** and **T**. When a lambda-nospread function is applied to some arguments, each of these arguments is evaluated and the values stored on the stack. *VAR* is then bound to the *number* of arguments which have been evaluated. For example, if **FOO** is defined by **(LAMBDA X ...)**, when **(FOO A B C)** is evaluated, **A**, **B**, and **C** are evaluated and **X** is bound to 3. *VAR* should *never* be reset.

The following functions are used for accessing the arguments of lambda-nospread functions:

(ARG *VAR M***)** [NLambda Function]

Returns the *M*th argument for the lambda-nospread function whose argument list is *VAR*. *VAR* is the *name* of the atomic argument list to a lambda-nospread function, and is not evaluated; *M* is the number of the desired argument, and is evaluated. The value of **ARG** is undefined for *M* less than or equal to 0 or greater than the *value* of *VAR*.

(SETARG *VAR M X***)** [NLambda Function]

Sets the *M*th argument for the lambda-nospread function whose argument list is *VAR* to *X*. *VAR* is not evaluated; *M* and *X* are evaluated. *M* should be between 1 and the value of *VAR*.

In the example below, the function **FOO** is defined to collect and return a list of all of the evaluated arguments it is given (the value of the **for** statement).

```
←(DEFINEQ (FOO
   (LAMBDA X
     (for ARGNUM from 1 to X collect (ARG X ARGNUM)]
```

```
(FOO)
←(FOO 99 (PLUS 3 4))
(99 7)
←(FOO 99 (PLUS 3 4) (TIMES 3 4))
(99 7 12)
```

10.1.4 Nlambda-Nospread Functions

Nlambda-nospread functions take a variable number of unevaluated arguments. An nlambda-nospread expr definition has the form:

(NLAMBDA *VAR* $FORM_1$ ... $FORM_N$**)**

VAR may be any litatom, except **NIL** and **T**. Though similar in form to lambda-nospread expr definitions, an nlambda-nospread is evaluated quite differently. When an nlambda-nospread function is applied to some arguments, *VAR* is simply bound to a list of the unevaluated arguments. The user may pick apart this list, and evaluate different arguments.

In the example below, **FOO** is defined to return the reverse of the list of arguments it is given (unevaluated):

```
←(DEFINEQ (FOO (NLAMBDA X (REVERSE X))))
(FOO)
←(FOO 99 (PLUS 3 4))
((PLUS 3 4) 99)
←(FOO 99 (PLUS 3 4) (TIMES 3 4))
((TIMES 3 4) (PLUS 3 4) 99)
```

Note: The warning about evaluating arguments to nlambda functions (page 10.5) also applies to nlambda-nospread function.

10.1.5 Compiled Functions

Functions defined by expr definitions can be compiled by the Interlisp compiler (page 18.1). The compiler produces compiled code objects (of data type **CCODEP**) which execute more quickly than the corresponding expr definition code. Functions defined by compiled code objects may have the same four types as expr definitions (lambda/nolambda, spread/nospread). Functions created by the compiler are referred to as compiled functions.

10.1.6 Function Type Functions

There are a variety of functions used for examining the type, argument list, etc. of functions. These functions may be given either a litatom, in which case they obtain the function definition from the litatom's definition cell, or a function definition itself.

(FNTYP *FN*) [Function]

Returns **NIL** if *FN* is not a function definition or the name of a defined function. Otherwise **FNTYP** returns one of the following litatoms, depending on the type of function definition:

EXPR Lambda-spread expr definition.

CEXPR Lambda-spread compiled definition.

FEXPR Nlambda-spread expr definition.

CFEXPR Nlambda-spread compiled definition.

EXPR* Lambda-nospread expr definition.

CEXPR* Lambda-nospread compiled definition.

FEXPR* Nlambda-nospread expr definition.

CFEXPR* Nlambda-nospread compiled definition.

FUNARG **FNTYP** returns the litatom **FUNARG** if *FN* is a **FUNARG** expression. See page 10.18.

EXPR, FEXPR, EXPR*, and **FEXPR*** indicate that *FN* is defined by an expr definition. **CEXPR, CFEXPR, CEXPR*,** and **CFEXPR*** indicate that *FN* is defined by a compiled definition, as indicated by the prefix **C**. The suffix * indicates that *FN* has an indefinite number of arguments, i.e., is a nospread functions. The prefix **F** indicates unevaluated arguments. Thus, for example, a **CFEXPR*** is a compiled nospread-nlambda function.

(EXPRP *FN*) [Function]

Returns **T** if **(FNTYP *FN*)** is either **EXPR, FEXPR, EXPR*,** or **FEXPR***; **NIL** otherwise. However, **(EXPRP *FN*)** is also true if *FN* is (has) a list definition, even if it does not begin with **LAMBDA** or **NLAMBDA**. In other words, **EXPRP** is not quite as selective as **FNTYP**.

(CCODEP *FN*) [Function]

Returns **T** if **(FNTYP *FN*)** is either **CEXPR, CFEXPR, CEXPR*,** or **CFEXPR***; **NIL** otherwise.

(ARGTYPE *FN*) [Function]

FN is the name of a function or its definition. **ARGTYPE** returns 0, 1, 2, or 3, or **NIL** if *FN* is not a function. The interpretation of this value is:

0 Lambda-spread function (**EXPR, CEXPR**)

1 Nlambda-spread function (**FEXPR, CFEXPR**)

2 Lambda-nospread function (**EXPR*, CEXPR***)

3 Nlambda-nospread function (**FEXPR***, **CFEXPR***)

i.e., **ARGTYPE** corresponds to the *rows* of **FNTYP**'s.

(NARGS *FN***)** [Function]

Returns the number of arguments of *FN*, or **NIL** if *FN* is not a function. If *FN* is a nospread function, the value of **NARGS** is 1.

(ARGLIST *FN***)** [Function]

Returns the "argument list" for *FN*. Note that the "argument list" is a litatom for nospread functions. Since **NIL** is a possible value for **ARGLIST**, an error is generated, **ARGS NOT AVAILABLE**, if *FN* is not a function.

If *FN* is a compiled function, the argument list is constructed, i.e., each call to **ARGLIST** requires making a new list. For functions defined by expr definitions, lists beginning with **LAMBDA** or **NLAMBDA**, the argument list is simply **CADR** of **GETD**. If *FN* has an expr definition, and **CAR** of the definition is not **LAMBDA** or **NLAMBDA**, **ARGLIST** will check to see if **CAR** of the definition is a member of **LAMBDASPLST** (page 20.14). If it is, **ARGLIST** presumes this is a function object the user is defining via **DWIMUSERFORMS** (page 20.11), and simply returns **CADR** of the definition as its argument list. Otherwise **ARGLIST** generates an error as described above.

(SMARTARGLIST *FN EXPLAINFLG TAIL***)** [Function]

A "smart" version of **ARGLIST** that tries various strategies to get the arglist of *FN*.

First, **SMARTARGLIST** checks the property list of *FN* under the property **ARGNAMES**. For spread functions, the argument list itself is stored. For nospread functions, the form is **(NIL** $ARGLIST_1$. $ARGLIST_2$**)**, where $ARGLIST_1$ is the value **SMARTARGLIST** should return when *EXPLAINFLG* = **T**, and $ARGLIST_2$ the value when *EXPLAINFLG* = **NIL**. For example, **(GETPROP 'DEFINEQ 'ARGNAMES) = (NIL (X1 XI ... XN) . X)**. This allows the user to specify special argument lists.

Second, if *FN* is not defined as a function, **SMARTARGLIST** attempts spelling correction on *FN* by calling **FNCHECK** (page 20.23), passing *TAIL* to be used for the call to **FIXSPELL**. If unsuccessful, an error will be generated, *FN* **NOT A FUNCTION**.

Third, if *FN* is known to the file package (page 17.1) but not loaded in, **SMARTARGLIST** will obtain the arglist information from the file.

Otherwise, **SMARTARGLIST** simply returns **(ARGLIST** *FN***)**.

SMARTARGLIST is used by **BREAK** (page 15.5) and **ADVISE** (page 15.11) with *EXPLAINFLG* = **NIL** for constructing equivalent expr definitions, and by the TTYIN in-line command **?=** (page 26.33), with *EXPLAINFLG* = **T**.

10.2 Defining Functions

Function definitions are stored in a "function definition cell" associated with each litatom. This cell is directly accessible via the two functions **PUTD** and **GETD** (page 10.11), but it is usually easier to define functions with **DEFINEQ**:

(DEFINEQ X_1 X_2 ... X_N**)** [NLambda NoSpread Function]

DEFINEQ is the function normally used for defining functions. It takes an indefinite number of arguments which are not evaluated. Each X_i must be a list defining one function, of the form **(***NAME DEFINITION***)**. For example:

(DEFINEQ (DOUBLE (LAMBDA (X) (IPLUS X X))))

The above expression will define the function **DOUBLE** with the expr definition **(LAMBDA (X) (IPLUS X X))**. X_i may also have the form **(***NAME ARGS* . *DEF-BODY***)**, in which case an appropriate lambda expr definition will be constructed. Therefore, the above expression is exactly the same as:

(DEFINEQ (DOUBLE (X) (IPLUS X X)))

Note that this alternate form can only be used for Lambda functions. The first form must be used to define an nlambda function.

DEFINEQ returns a list of the names of the functions defined.

(DEFINE *X* **—)** [Function]

Lambda-spread version of **DEFINEQ**. Each element of the list *X* is itself a list either of the form **(***NAME DEFINITION***)** or **(***NAME ARGS* . *DEF-BODY***)**. **DEFINE** will generate an error, **INCORRECT DEFINING FORM**, on encountering an atom where a defining list is expected.

Note: **DEFINE** and **DEFINEQ** will operate correctly if the function is already defined and **BROKEN**, **ADVISED**, or **BROKEN-IN**.

For expressions involving type-in only, if the time stamp facility is enabled (page 16.76), both **DEFINE** and **DEFINEQ** will stamp the definition with the user's initials and date.

UNSAFE.TO.MODIFY.FNS [Variable]

Value is a list of functions that should not be redefined, because doing so may cause unusual bugs (or crash the system!). If the user tries to modify a function on this list (using **DEFINEQ**, **TRACE**, etc), the system will print "**Warning:** *XXX* **may be safe to modify -- continue?**" If the users types "Yes", the function is modified, otherwise an error occurs. This provides a measure of safety for novices who may accidently redefine important system functions. Users can add their own functions onto this list.

Note: By convention, all functions starting with the character backslash ("\") are system internal functions, which should never be redefined or modified by the user. Backslash functions are not on **UNSAFE.TO.MODIFY.FNS**, so trying to redefine them will not cause a warning.

DFNFLG [Variable]

DFNFLG is a global variable that effects the operation of **DEFINEQ** and **DEFINE**. If **DFNFLG = NIL**, an attempt to *redefine* a function *FN* will cause **DEFINE** to print the message (*FN* **REDEFINED)** and to save the old definition of *FN* using **SAVEDEF** (page 17.27) before redefining it (except if the old and new definitions are **EQUAL**, inwhich case the effect is simply a no-op). If **DFNFLG = T**, the function is simply redefined. If **DFNFLG = PROP** or **ALLPROP**, the new definition is stored on the property list under the property **EXPR**. **ALLPROP** also affects the operation of **RPAQQ** and **RPAQ** (page 17.54). **DFNFLG** is initially **NIL**.

DFNFLG is reset by **LOAD** (page 17.6) to enable various ways of handling the defining of functions and setting of variables when loading a file. For most applications, the user will not reset **DFNFLG** directly.

Note: The compiler does NOT respect the value of **DFNFLG** when it redefines functions to their compiled definitions (see page 18.1). Therefore, if you set **DFNFLG** to **PROP** to completely avoid inadvertantly redefining something in your running system, you *must* use compile mode **F**, not **ST**.

Note: The functions **SAVEDEF** and **UNSAVEDEF** (page 17.27) can be useful for "saving" and restoring function definitions from property lists.

(GETD *FN*) [Function]

Returns the function definition of *FN*. Returns **NIL** if *FN* is not a litatom, or has no definition.

GETD of a compiled function constructs a pointer to the definition, with the result that two successive calls do not necessarily produce **EQ** results. **EQP** or **EQUAL** must be used to compare compiled definitions.

(PUTD *FN DEF* **—)** [Function]

Puts *DEF* into *FN*'s function cell, and returns *DEF*. Generates an error, **ARG NOT LITATOM,** if *FN* is not a litatom. Generates an error, **ILLEGAL ARG,** if *DEF* is a string, number, or a litatom other than **NIL.**

(MOVD *FROM TO COPYFLG* **—)** [Function]

Moves the definition of *FROM* to *TO*, i.e., redefines *TO*. If *COPYFLG* = **T**, a **COPY** of the definition of *FROM* is used. *COPYFLG* = **T** is only meaningful for expr definitions, although **MOVD** works for compiled functions as well. **MOVD** returns *TO*.

COPYDEF (page 17.27) is a higher-level function that only moves expr definitions, but works also for variables, records, etc.

(MOVD? *FROM TO COPYFLG* **—)** [Function]

If *TO* is not defined, same as **(MOVD** *FROM TO COPYFLG***)**. Otherwise, does nothing and returns **NIL.**

10.3 Function Evaluation

Usually, function application is done automatically by the Interlisp interpreter. If a form is typed into Interlisp whose **CAR** is a function, this function is applied to the arguments in the **CDR** of the form. These arguments are evaluated or not, and bound to the function parameters, as determined by the type of the function, and the body of the function is evaluated. This sequence is repeated as each form in the body of the function is evaluated.

There are some situations where it is necessary to explicitly call the evaluator, and Interlisp supplies a number of functions that will do this. These functions take "functional arguments", which may either be litatoms with function definitions, or expr definition forms such as **(LAMBDA (X) ...)**, or **FUNARG** expressions (see page 10.18).

(APPLY *FN ARGLIST* **—)** [Function]

Applies the function *FN* to the arguments in the list *ARGLIST*, and returns its value. **APPLY** is a lambda function, so its arguments are evaluated, but the individual elements of *ARGLIST* are not evaluated. Therefore, lambda and nlambda functions are treated the same by **APPLY**—lambda functions take their arguments from *ARGLIST* without evaluating them. For example:

←(APPLY 'APPEND '((PLUS 1 2 3) (4 5 6)))
(PLUS 1 2 3 4 5 6)

Note that *FN* may explicitly evaluate one or more of its arguments itself. For example, the system function **SETQ** is an nlambda function that explicitly evaluates its second argument. Therefore, **(APPLY 'SETQ '(FOO (ADD1 3)))** will set **FOO** to 4, instead of setting it to the expression **(ADD1 3)**.

APPLY can be used for manipulating expr definitions, for example:

←(APPLY '(LAMBDA (X Y) (ITIMES X Y)) '(3 4))
12

(APPLY* ***FN*** ***ARG*** $_1$ ***ARG*** $_2$ **...** ***ARG*** $_N$ **)** [NoSpread Function]

Nospread version of **APPLY**. Applies the function *FN* to the arguments *ARG*$_1$ *ARG*$_2$ ... *ARG*$_N$. For example,

←(APPLY* 'APPEND '(PLUS 1 2 3) '(4 5 6))
(PLUS 1 2 3 4 5 6)

(EVAL ***X*** **—)** [Function]

EVAL evaluates the expression *X* and returns this value, i.e., **EVAL** provides a way of calling the Interlisp interpreter. Note that **EVAL** is itself a lambda function, so *its* argument is first evaluated, e.g.,

←(SETQ FOO '(ADD1 3))
(ADD1 3)
←(EVAL FOO)
4
←(EVAL 'FOO)
(ADD1 3)

(QUOTE ***X*****)** [NLambda NoSpread Function]

QUOTE prevents its arguments from being evaluated. Its value is *X* itself, e.g., **(QUOTE FOO)** is **FOO**.

Interlisp functions can either evaluate or not evaluate their arguments. **QUOTE** can be used in those cases where it is desirable to specify arguments unevaluated.

Note: The character single-quote (') is defined with a read macro so it returns the next expression, wrapped in a call to **QUOTE** (page 25.42). For example, **'FOO** reads as **(QUOTE FOO)**. This is the form used for examples in this manual.

Since giving **QUOTE** more than one argument is almost always a parentheses error, and one that would otherwise go undetected,

QUOTE itself generates an error in this case, **PARENTHESIS ERROR**.

(KWOTE *X***)** [Function]

Value is an expression which when evaluated yields *X*. If *X* is **NIL** or a number, this is *X* itself. Otherwise, **(LIST (QUOTE QUOTE)** *X***)**. For example,

(KWOTE 5) => 5

(KWOTE (CONS 'A 'B)) => (QUOTE (A . B))

(NLAMBDA.ARGS *X***)** [Function]

This function interprets its argument as a list of unevaluated Nlambda arguments. If any of the elements in this list are of the form **(QUOTE ...)**, the enclosing **QUOTE** is stripped off. Actually, **NLAMBDA.ARGS** stops processing the list after the first non-quoted argument. Therefore, whereas **(NLAMBDA.ARGS '((QUOTE FOO) BAR)) -> (FOO BAR)**, **(NLAMBDA.ARGS '(FOO (QUOTE BAR))) -> (FOO (QUOTE BAR))**.

NLAMBDA.ARGS is called by a number of nlambda functions in the system, to interpret their arguments. For instance, the function **BREAK** calls **NLAMBDA.ARGS** so that **(BREAK 'FOO)** will break the function **FOO**, rather than the function **QUOTE**.

(EVALA *X A***)** [Function]

Simulates association list variable lookup. *X* is a form, *A* is a list of the form:

(($NAME_1$. VAL_1) ($NAME_2$. VAL_2) ... ($NAME_N$. VAL_N))

The variable names and values in *A* are "spread" on the stack, and then *X* is evaluated. Therefore, any variables appearing free in *X*, that also appears as **CAR** of an element of *A* will be given the value in the **CDR** of that element.

(DEFEVAL *TYPE FN***)** [Function]

Specifies how a datum of a particular type is to be evaluated. Intended primarily for user defined data types, but works for all data types except lists, literal atoms, and numbers. *TYPE* is a type name. *FN* is a function object, i.e. name of a function or a lambda expression. Whenever the interpreter encounters a datum of the indicated type, *FN* is applied to the datum and its value returned as the result of the evaluation. **DEFEVAL** returns the previous evaling function for this type. If *FN* = **NIL**, **DEFEVAL** returns the current evaling function without changing it. If *FN* = **T**, the evaling function is set back to the system default (which for all data types except lists is to return the datum itself).

Note: **COMPILETYPELST** (page 18.11) permits the user to specify how a datum of a particular type is to be compiled.

(EVALHOOK *FORM EVALHOOKFN***)** [Function]

EVALHOOK evaluates the expression *FORM*, and returns its value. While evaluating *FORM*, the function **EVAL** behaves in a special way. Whenever a list other than *FORM* itself is to be evaluated, whether implicitly or via an explicit call to **EVAL**, *EVALHOOKFN* is invoked (it should be a function), with the form to be evaluated as its argument. *EVALHOOKFN* is then responsible for evaluating the form; whatever is returned is assumed to be the result of evaluating the form. During the execution of *EVALHOOKFN*, this special evaluation is turned off. (Note that **EVALHOOK** does not effect the evaluations of variables, only of lists).

Here is an example of a simple tracing routine that uses the **EVALHOOK** feature:

```
←(DEFINEQ (PRINTHOOK (FORM)
   (printout T "eval: " FORM T)
   (EVALHOOK FORM (FUNCTION PRINTHOOK]
(PRINTHOOK)
```

Using **PRINTHOOK**, one might see the following interaction:

```
←(EVALHOOK '(LIST (CONS 1 2) (CONS 3 4)) 'PRINTHOOK)
eval: (CONS 1 2)
eval: (CONS 3 4)
((1 . 2) (3 . 4))
```

10.4 Iterating and Mapping Functions

The functions below are used to evaluate a form or apply a function repeatedly. **RPT**, **RPTQ**, and **FRPTQ** evaluate an expression a specified number of times. **MAP**, **MAPCAR**, **MAPLIST**, etc. apply a given function repeatedly to different elements of a list, possibly constructing another list.

These functions allow efficient iterative computations, but they are difficult to use. For programming iterative computations, it is usually better to use the CLISP Iterative Statement facility (page 9.9), which provides a more general and complete facility for expressing iterative statements. Whenever possible, CLISP translates iterative statements into expressions using the functions below, so there is no efficiency loss.

(RPT *N FORM***)** [Function]

Evaluates the expression *FORM*, *N* times. Returns the value of the last evaluation. If *N* less than or equal to 0, *FORM* is not evaluated, and **RPT** returns **NIL**.

Before each evaluation, the local variable **RPTN** is bound to the number of evaluations yet to take place. This variable can be referenced within *FORM*. For example, **(RPT 10 '(PRINT RPTN))** will print the numbers 10, 9, ... 1, and return 1.

(RPTQ $N\ FORM_1\ FORM_2 \ldots FORM_N$**)** [NLambda NoSpread Function]

Nlambda-nospread version of **RPT**: *N* is evaluated, $FORM_i$ are not. Returns the value of the last evaluation of $FORM_N$.

(FRPTQ $N\ FORM_1\ FORM_2 \ldots FORM_N$**)** [NLambda NoSpread Function]

Faster version of **RPTQ**. Does not bind **RPTN**.

(MAP *MAPX MAPFN1 MAPFN2***)** [Function]

If *MAPFN2* is **NIL**, **MAP** applies the function *MAPFN1* to successive tails of the list *MAPX*. That is, first it computes **(***MAPFN1 MAPX***)**, and then **(***MAPFN1* **(CDR** *MAPX***))**, etc., until *MAPX* becomes a non-list. If *MAPFN2* is provided, **(***MAPFN2 MAPX***)** is used instead of **(CDR** *MAPX***)** for the next call for *MAPFN1*, e.g., if *MAPFN2* were **CDDR**, alternate elements of the list would be skipped. **MAP** returns **NIL**.

(MAPC *MAPX MAPFN1 MAPFN2***)** [Function]

Identical to **MAP**, except that **(***MAPFN1* **(CAR** *MAPX***))** is computed at each iteration instead of **(***MAPFN1 MAPX***)**, i.e., **MAPC** works on elements, **MAP** on tails. **MAPC** returns **NIL**.

(MAPLIST *MAPX MAPFN1 MAPFN2***)** [Function]

Successively computes the same values that **MAP** would compute, and returns a list consisting of those values.

(MAPCAR *MAPX MAPFN1 MAPFN2***)** [Function]

Computes the same values that **MAPC** would compute, and returns a list consisting of those values, e.g., **(MAPCAR X 'FNTYP)** is a list of **FNTYP**s for each element on **X**.

(MAPCON *MAPX MAPFN1 MAPFN2***)** [Function]

Computes the same values as **MAP** and **MAPLIST** but **NCONC**s these values to form a list which it returns.

(MAPCONC *MAPX MAPFN1 MAPFN2***)** [Function]

Computes the same values as **MAPC** and **MAPCAR**, but **NCONC**s the values to form a list which it returns.

Note that **MAPCAR** creates a new list which is a mapping of the old list in that each element of the new list is the result of applying a function to the corresponding element on the original list. **MAPCONC** is used when there are a *variable* number of elements (including none) to be inserted at each iteration. Examples:

```
(MAPCONC '(A B C NIL D NIL)
    '(LAMBDA (Y) (if (NULL Y) then NIL else (LIST Y))))
 => (A B C D)
```

This **MAPCONC** returns a list consisting of *MAPX* with all **NIL**s removed.

```
(MAPCONC '((A B) C (D E F) (G) H I)
    '(LAMBDA (Y) (if (LISTP Y) then Y else NIL)))
 => (A B D E F G)
```

This **MAPCONC** returns a linear list consisting of all the lists on *MAPX*.

Since **MAPCONC** uses **NCONC** to string the corresponding lists together, in this example the original list will be altered to be **((A B D E F G) C (D E F G) (G) H I)**. If this is an undesirable side effect, the functional argument to **MAPCONC** should return instead a top level copy of the lists, i.e. **(LAMBDA (Y) (if (LISTP Y) then (APPEND Y) else NIL)))**.

(MAP2C *MAPX MAPY MAPFN1 MAPFN2***)** [Function]

Identical to **MAPC** except *MAPFN1* is a function of two arguments, and **(***MAPFN1* **(CAR** *MAPX***) (CAR** *MAPY***))** is computed at each iteration. Terminates when either *MAPX* or *MAPY* is a non-list.

MAPFN2 is still a function of one argument, and is applied twice on each iteration; **(***MAPFN2 MAPX***)** gives the new *MAPX*, **(***MAPFN2 MAPY***)** the new *MAPY*. **CDR** is used if *MAPFN2* is not supplied, i.e., is **NIL**.

(MAP2CAR *MAPX MAPY MAPFN1 MAPFN2***)** [Function]

Identical to **MAPCAR** except *MAPFN1* is a function of two arguments and **(***MAPFN1* **(CAR** *MAPX***) (CAR** *MAPY***))** is used to assemble the new list. Terminates when either *MAPX* or *MAPY* is a non-list.

(SUBSET *MAPX MAPFN1 MAPFN2***)** [Function]

Applies *MAPFN1* to elements of *MAPX* and returns a list of those elements for which this application is non-**NIL**, e.g.,

(SUBSET '(A B 3 C 4) 'NUMBERP) = (3 4).

MAPFN2 plays the same role as with **MAP**, **MAPC**, et al.

(EVERY *EVERYX EVERYFN1 EVERYFN2***)** [Function]

Returns **T** if the result of applying *EVERYFN1* to each element in *EVERYX* is true, otherwise **NIL**. For example, **(EVERY '(X Y Z) 'ATOM) => T**.

EVERY operates by evaluating **(***EVERYFN1* **(CAR** *EVERYX***)** *EVERYX***)**. The second argument is passed to *EVERYFN1* so that it can look at the next element on *EVERYX* if necessary. If *EVERYFN1* yields **NIL**, **EVERY** immediately returns **NIL**. Otherwise, **EVERY** computes **(***EVERYFN2 EVERYX***)**, or **(CDR** *EVERYX***)** if *EVERYFN2* = **NIL**, and uses this as the "new" *EVERYX*, and the process continues. For example, **(EVERY** *X* **'ATOM 'CDDR)** is true if every *other* element of *X* is atomic.

(SOME *SOMEX SOMEFN1 SOMEFN2***)** [Function]

Returns the tail of *SOMEX* beginning with the first element that satisfies *SOMEFN1*, i.e., for which *SOMEFN1* applied to that element is true. Value is **NIL** if no such element exists. **(SOME X '(LAMBDA (Z) (EQUAL Z Y)))** is equivalent to **(MEMBER Y X)**. **SOME** operates analogously to **EVERY**. At each stage, **(***SOMEFN1* **(CAR** *SOMEX***)** *SOMEX***)** is computed, and if this is not **NIL**, *SOMEX* is returned as the value of **SOME**. Otherwise, **(***SOMEFN2 SOMEX***)** is computed, or **(CDR** *SOMEX***)** if *SOMEFN2* = **NIL**, and used for the next *SOMEX*.

(NOTANY *SOMEX SOMEFN1 SOMEFN2***)** [Function]

(NOT (SOME *SOMEX SOMEFN1 SOMEFN2***))**

(NOTEVERY *EVERYX EVERYFN1 EVERYFN2***)** [Function]

(NOT (EVERY *EVERYX EVERYFN1 EVERYFN2***))**

(MAPRINT *LST FILE LEFT RIGHT SEP PFN LISPXPRINTFLG***)** [Function]

A general printing function. For each element of the list *LST*, applies *PFN* to the element, and *FILE*. If *PFN* is **NIL**, **PRIN1** is used. Between each application, **MAPRINT** performs **PRIN1** of *SEP* (or " " if *SEP* = **NIL**). If *LEFT* is given, it is printed (using **PRIN1**) initially; if *RIGHT* is given it is printed (using **PRIN1**) at the end.

For example, **(MAPRINT X NIL '%('%))** is equivalent to **PRIN1** for lists. To print a list with commas between each element and a final "." one could use **(MAPRINT X T NIL '%. '%,)**.

If *LISPXPRINTFLG* = **T**, **LISPXPRIN1** (page 13.25) is used instead of **PRIN1**.

10.5 Functional Arguments

The functions that call the Interlisp-D evaluator take "functional arguments", which may either be litatoms with function definitions, or expr definition forms such as **(LAMBDA (X) ...)**, or **FUNARG** expressions (below).

The following functions are useful when one wants to supply a functional argument which will always return **NIL**, **T**, or 0. Note that the arguments X_1 ... X_N to these functions are evaluated, though they are not used.

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

Returns **NIL**.

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

Returns **T**.

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

Returns 0.

When using expr definitions as functional arguments, they should be enclosed within the function **FUNCTION** rather than **QUOTE**, so that they will be compiled as separate functions. **FUNCTION** can also be used to create **FUNARG** expressions, which can be used to solve some problems with referencing free variables, or to create functional arguments which carry "state" along with them.

(FUNCTION *FN ENV***)** [NLambda Function]

If *ENV* = **NIL**, **FUNCTION** is the same as **QUOTE**, except that it is treated differently when compiled. Consider the function definition:

```
(DEFINEQ (FOO (LST)
      (FIE LST (FUNCTION (LAMBDA (Z) (ITIMES Z Z))]
```

FOO calls the function **FIE** with the value of **LST** and the expr definition **(LAMBDA (Z) (LIST (CAR Z)))**.

If **FOO** is run interpreted, it doesn't make any difference whether **FUNCTION** or **QUOTE** is used. However, when **FOO** is compiled, if **FUNCTION** is used the compiler will define and compile the expr definition as an auxiliary function (See page 18.10). The compiled expr definition will run considerably faster, which can make a big difference if it is applied repeatedly.

Note: Compiling **FUNCTION** will *not* create an auxiliary function if it is a functional argument to a function that compiles open, such as most of the mapping functions (**MAPCAR**, **MAPLIST**, etc.).

If *ENV* is not **NIL**, it can be a list of variables that are (presumably) used freely by *FN*. In this case, the value of **FUNCTION** is an expression of the form **(FUNARG** *FN POS***)**, where *POS* is a stack pointer to a frame that contains the variable bindings for those variables on *ENV*. *ENV* can also be a stack pointer itself, in which case the value of **FUNCTION** is **(FUNARG** *FN ENV***)**. Finally, *ENV* can be an atom, in which case it is evaluated, and the value interpreted as described above.

As explained above, one of the possible values that **FUNCTION** can return is the form **(FUNARG** *FN POS***)**, where *FN* is a function and *POS* is a stack pointer. **FUNARG** is not a function itself. Like **LAMBDA** and **NLAMBDA**, it has meaning and is specially recognized by Interlisp only in the context of applying a function to arguments. In other words, the expression **(FUNARG** *FN POS***)** is used exactly like a function. When a **FUNARG** expression is applied or is **CAR** of a form being **EVAL**'ed, the **APPLY** or **EVAL** takes place in the access environment specified by *ENV* (see page 11.1). Consider the following example:

```
←(DEFINEQ (DO.TWICE (FN VAL)
      (APPLY* FN (APPLY* FN VAL))) )
(DO.TWICE)
←(DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X X))]
     5)
20
←(SETQ VAL 1)
1
←(DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X VAL))]
     5)
15
←(DO.TWICE [FUNCTION (LAMBDA (X) (IPLUS X VAL)) (VAL)]
     5)
7
```

DO.TWICE is defined to apply a function **FN** to a value **VAL**, and apply **FN** again to the value returned; in other words it calculates **(FN (FN VAL))**. Given the expr definition **(LAMBDA (X) (IPLUS X X))**, which doubles a given value, it correctly calculates **(FN (FN 5))** = **(FN 10)** = 20. However, when given **(LAMBDA (X) (IPLUS X VAL))**, which should add the value of the global variable **VAL** to the argument **X**, it does something unexpected, returning 15, rather than 5 + 1 + 1 = 7. The problem is that when the expr definition is evaluated, it is evaluated in the context of **DO.TWICE**, where **VAL** is bound to the second argument of **DO.TWICE**, namely 5. In this case, one solution is to use the *ENV* argument to **FUNCTION** to construct a **FUNARG** expression which contains the value of **VAL** at the time that the **FUNCTION** is executed. Now, when **(LAMBDA (X) (IPLUS X VAL))** is evaluated, it is evaluated in an environment where the global value of **VAL** is accessable. Admittedly, this is a somewhat contrived example (it would be easy enough to change the argument names to **DO.TWICE** so there would be no conflict), but this situation arises occasionally with large systems of programs that construct functions, and pass them around.

Note: System functions with functional arguments (**APPLY**, **MAPCAR**, etc.) are compiled so that their arguments are local, and not accessable (see page 18.5). This reduces problems with conflicts with free variables used in functional arguments.

FUNARG expressions can be used for more than just circumventing the clashing of variables. For example, a **FUNARG** expression can be returned as the value of a computation, and then used "higher up". Furthermore, if the function in a **FUNARG** expression *sets* any of the variables contained in the frame, only the frame would be changed. For example, consider the following function:

```
←(DEFINEQ (MAKECOUNTER (CNT)
  (FUNCTION [LAMBDA NIL
        (PROG1 CNT (SETQ CNT (ADD1 CNT]
       (CNT)))]
```

The function **MAKECOUNTER** returns a **FUNARG** that increments and returns the previous value of the counter **CNT**. However, this is done within the environment of the call to **MAKECOUNTER** where **FUNCTION** was executed, which the **FUNARG** expression "carries around" with it, even after **MAKECOUNTER** has finished executing. Note that each call to **MAKECOUNTER** creates a **FUNARG** expression with a new, independent environment, so that multiple counters can be generated and used:

```
←(SETQ C1 (MAKECOUNTER 1))
(FUNARG (LAMBDA NIL (PROG1 CNT (SETQ CNT (ADD1 CNT))))
#1,13724/*FUNARG)
←(APPLY C1)
1
←(APPLY C1)
2
←(SETQ C2 (MAKECOUNTER 17))
(FUNARG (LAMBDA NIL (PROG1 CNT (SETQ CNT (ADD1 CNT))))
#1,13736/*FUNARG)
←(APPLY C2)
17
```

←(APPLY C2)
18
←(APPLY C1)
3
←(APPLY C2)
19

By creating a **FUNARG** expression with **FUNCTION**, a program can create a function object which has updateable binding(s) associated with the object which last *between* calls to it, but are only accessible through that instance of the function. For example, using the **FUNARG** device, a program could maintain two different instances of the same random number generator in different states, and run them independently.

10.6 Macros

Macros provide an alternative way of specifying the action of a function. Whereas function definitions are evaluated with a "function call", which involves binding variables and other housekeeping tasks, macros are evaluated by *translating* one Interlisp form into another, which is then evaluated.

A litatom may have both a function definition and a macro definition. When a form is evaluated by the interpreter, if the **CAR** has a function definition, it is used (with a function call), otherwise if it has a macro definition, then that is used. However, when a form is compiled, the **CAR** is checked for a macro definition first, and only if there isn't one is the function definition compiled. This allows functions that behave differently when compiled and interpreted. For example, it is possible to define a function that, when interpreted, has a function definition that is slow and has a lot of error checks, for use when debugging a system. This function could also have a macro definition that defines a fast version of the function, which is used when the debugged system is compiled.

Macro definitions are represented by lists that are stored on the property list of a litatom. Macros are often used for functions that should be compiled differently in different Interlisp implementations, and the exact property name a macro definition is stored under determines whether it should be used in a particular implementation. The global variable **MACROPROPS** contains a list of all possible macro property names which should be saved by the **MACROS** file package command. Typical macro property names are **DMACRO** for Interlisp-D, **10MACRO** for Interlisp-10, **VAXMACRO** for Interlisp-VAX, **JMACRO** for Interlisp-Jerico, and **MACRO** for "implementation independent" macros. The global variable **COMPILERMACROPROPS** is a list of macro property names. Interlisp determines whether a litatom has a macro definition by checking these property names, in order, and using the first non-**NIL** property value as the macro definition. In Interlisp-D this list contains **DMACRO** and **MACRO** in that order so that **DMACRO**s will override the implementation-independent **MACRO** properties. In general, use a **DMACRO** property for macros that are to be used only in Interlisp-D, use **10MACRO** for macros that are to be used only in Interlisp-10, and use **MACRO** for macros that are to affect both systems.

Macro definitions can take the following forms:

(LAMBDA ...)
(NLAMBDA ...)

A function can be made to compile open by giving it a macro definition of the form **(LAMBDA ...)** or **(NLAMBDA ...)**, e.g., **(LAMBDA (X) (COND ((GREATERP X 0) X) (T (MINUS X))))** for **ABS**. The effect is as if the macro definition were written in place of the function wherever it appears in a function being compiled, i.e., it compiles as a lambda or nlambda expression. This saves the time necessary to call the function at the price of more compiled code generated in-line.

(NIL *EXPRESSION*)
(*LIST EXPRESSION*)

"Substitution" macro. Each argument in the form being evaluated or compiled is substituted for the corresponding atom in *LIST*, and the result of the substitution is used instead of the form. For example, if the macro definition of **ADD1** is **((X) (IPLUS X 1))**, then, **(ADD1 (CAR Y))** is compiled as **(IPLUS (CAR Y) 1)**.

Note that **ABS** could be defined by the substitution macro **((X) (COND ((GREATERP X 0) X) (T (MINUS X))))**. In this case, however, **(ABS (FOO X))** would compile as

```
(COND ((GREATERP (FOO X) 0)
   (FOO X))
  (T (MINUS (FOO X))))
```

and **(FOO X)** would be evaluated two times. (Code to evaluate **(FOO X)** would be generated three times.)

(OPENLAMBDA *ARGS BODY*)

This is a cross between substitution and **LAMBDA** macros. When the compiler processes an **OPENLAMBDA**, it attempts to substitute the actual arguments for the formals wherever this preserves the frequency and order of evaluation that would have resulted from a **LAMBDA** expression, and produces a **LAMBDA** binding only for those that require it.

Note: **OPENLAMBDA** assumes that it can substitute literally the actual arguments for the formal arguments in the body of the macro if the actual is side-effect free or a constant. Thus, you should be careful to use names in *ARGS* which don't occur in

BODY (except as variable references). For example, if **FOO** has a macro definition of

(OPENLAMBDA (ENV) (FETCH (MY-RECORD-TYPE ENV) OF BAR))

then **(FOO NIL)** will expand to

(FETCH (MY-RECORD-TYPE NIL) OF BAR)

T When a macro definition is the atom **T**, it means that the compiler should ignore the macro, and compile the function definition; this is a simple way of turning off other macros. For example, the user may have a function that runs in both Interlisp-D and Interlisp-10, but has a macro definition that should only be used when compiling in Interlisp-10. If the **MACRO** property has the macro specification, a **DMACRO** of **T** will cause it to be ignored by the Interlisp-D compiler. Note that this **DMACRO** would not be necessary if the macro were specified by a **10MACRO** instead of a **MACRO**.

(= . ***OTHER-FUNCTION*****)** A simple way to tell the compiler to compile one function exactly as it would compile another. For example, when compiling in Interlisp-D, **FRPLACA**s are treated as **RPLACA**s. This is achieved by having **FRPLACA** have a **DMACRO** of **(= . RPLACA)**.

(***LITATOM EXPRESSION*****)** If a macro definition begins with a litatom other than those given above, this allows *computation* of the Interlisp expression to be evaluated or compiled in place of the form. *LITATOM* is bound to the **CDR** of the calling form, *EXPRESSION* is evaluated, and the result of this evaluation is evaluated or compiled in place of the form. For example, **LIST** could be compiled using the computed macro:

```
[X (LIST 'CONS
     (CAR X)
     (AND (CDR X)
       (CONS 'LIST
         (CDR X]
```

This would cause **(LIST X Y Z)** to compile as **(CONS X (CONS Y (CONS Z NIL)))**. Note the recursion in the macro expansion.

If the result of the evaluation is the litatom **IGNOREMACRO**, the macro is ignored and the compilation of the expression proceeds as if there were no macro definition. If the litatom in question is normally treated specially by the compiler (**CAR**, **CDR**, **COND**, **AND**, etc.), and also has a macro, if the macro expansion returns **IGNOREMACRO**, the litatom will still be treated specially.

In Interlisp-10, if the result of the evaluation is the atom **INSTRUCTIONS**, no code will be generated by the compiler. It is then assumed the evaluation was done for effect and the necessary code, if any, has been added. This is a way of giving direct instructions to the compiler if you understand it.

Note: It is often useful, when constructing complex macro expressions, to use the **BQUOTE** facility (see page 25.42).

The following function is quite useful for debugging macro definitions:

(EXPANDMACRO *EXP QUIETFLG* **— —)** [Function]

Takes a form whose **CAR** has a macro definition and expands the form as it would be compiled. The result is prettyprinted, unless *QUIETFLG* = **T**, in which case the result is simply returned.

10.6.1 DEFMACRO

Macros defined with the function **DEFMACRO** are much like "computed" macros (page 10.23), in that they are defined with a form that is evaluated, and the result of the evaluation is used (evaluated or compiled) in place of the macro call. However, **DEFMACRO** macros support complex argument lists with optional arguments, default values, and keyword arguments. In addition, argument list destructuring is supported.

(DEFMACRO *NAME ARGS FORM***)** [NLambda NoSpread Function]

Defines *NAME* as a macro with the arguments *ARGS* and the definition form *FORM* (*NAME*, *ARGS*, and *FORM* are unevaluated). If an expression starting with *NAME* is evaluated or compiled, arguments are bound according to *ARGS*, *FORM* is evaluated, and the value of *FORM* is evaluated or compiled instead. The interpretation of *ARGS* is described below.

Note: Unlike the function **DEFMACRO** in Common Lisp, this function currently does not remove any function definition for *NAME*.

ARGS is a list that defines how the argument list passed to the macro *NAME* is interpreted. Specifically, *ARGS* defines a set of variables that are set to various arguments in the macro call (unevaluated), that *FORM* can reference to construct the macro form.

In the simplest case, *ARGS* is a simple list of variable names that are set to the corresponding elements of the macro call (unevaluated). For example, given:

(DEFMACRO FOO (A B) (LIST 'PLUS A B B))

The macro call **(FOO X (BAR Y Z))** will expand to **(PLUS X (BAR Y Z) (BAR Y Z))**.

The list *ARGS* can include any of a number of special "&-keywords" (beginning with the character "&") that are used to set variables to particular items from the macro call form, as follows:

&OPTIONAL Used to define optional arguments, possibly with default values. Each element on *ARGS* after **&OPTIONAL** until the next &-keyword or the end of the list defines an optional argument, which can either be a litatom or a list, interpreted as follows:

VAR

If an optional argument is specified as a litatom, that variable is set to the corresponding element of the macro call (unevaluated).

(*VAR DEFAULT***)**

If an optional argument is specified as a two element list, *VAR* is the variable to be set, and *DEFAULT* is a form that is evaluated and used as the default if there is no corresponding element in the macro call.

(*VAR DEFAULT VARSETP***)**

If an optional argument is specified as a three element list, *VAR* and *DEFAULT* are the variable to be set and the default form, and *VARSETP* is a variable that is set to **T** if the optional argument is given in the macro call, **NIL** otherwise. This can be used to determine whether the argument was not given, or whether it was specified with the default value.

For example, after

(DEFMACRO FOO (&OPTIONAL A (B 5) (C 6 CSET)) *FORM***)**

expanding the macro call **(FOO)** would cause *FORM* to be evaluated with **A** set to **NIL**, **B** set to **5**, **C** set to **6**, and **CSET** set to **NIL**. **(FOO 4 5 6)** would be the same, except that **A** would be set to **4** and **CSET** would be set to **T**.

&REST
&BODY Used to get a list of all additional arguments from the macro call. Either **&REST** or **&BODY** should be followed by a single litatom, which is set to a list of all arguments to the macro after the position of the &-keyword. For example, given

(DEFMACRO FOO (A B &REST C) *FORM***)**

expanding the macro call **(FOO 1 2 3 4 5)** would cause *FORM* to be evaluated with **A** set to **1**, **B** set to **2**, and **C** set to **(3 4 5)**.

Note: If the macro calling form contains keyword arguments (see **&KEY** below) these are included in the **&REST** list.

&KEY Used to define keyword arguments, that are specified in the macro call by including a "keyword" (a litatom starting with the character ":") followed by a value.

Each element on *ARGS* after **&KEY** until the next &-keyword or the end of the list defines a keyword argument, which can either be a litatom or a list, interpreted as follows:

VAR

(*VAR*)

((*KEYWORD VAR*))

If a keyword argument is specified by a single litatom *VAR*, or a one-element list containing *VAR*, it is set to the value of a keyword argument, where the keyword used is created by adding the character ":" to the front of *VAR*. If a keyword argument is specified by a single-element list containing a two-element list, *KEYWORD* is interpreted as the keyword (which should start with the letter ":"), and *VAR* is the variable to set.

(*VAR DEFAULT*)

((*KEYWORD VAR*) *DEFAULT*)

(*VAR DEFAULT VARSETP*)

((*KEYWORD VAR*) *DEFAULT VARSETP*)

If a keyword argument is specified by a two or three-element list, the first element of the list specifies the keyword and variable to set as above. Similar to **&OPTIONAL** (above), the second element *DEFAULT* is a form that is evaluated and used as the default if there is no corresponding element in the macro call, and the third element *VARSETP* is a variable that is set to **T** if the optional argument is given in the macro call, **NIL** otherwise.

For example, the form

(DEFMACRO FOO (&KEY A (B 5 BSET) ((:BAR C) 6 CSET)) *FORM***)**

Defines a macro with keys **:A**, **:B** (defaulting to 5), and **:BAR**. Expanding the macro call **(FOO :BAR 2 :A 1)** would cause *FORM* to be evaluated with **A** set to **1**, **B** set to **5**, **BSET** set to **NIL**, **C** set to 2, and **CSET** set to **T**.

&ALLOW-OTHER-KEYS It is an error for any keywords to be suplied in a macro call that are not defined as keywords in the macro argument list, unless either the &-keyword **&ALLOW-OTHER-KEYS** appears in *ARGS*, or the keyword **:ALLOW-OTHER-KEYS** (with a non-**NIL** value) appears in the macro call.

&AUX Used to bind and initialize auxiliary varables, using a syntax similar to **PROG** (page 9.8). Any elements after **&AUX** should be either litatoms or lists, interpreted as follows:

VAR

Single litatoms are interpreted as auxiliary variables that are initially bound to **NIL**.

(*VAR EXP*)

If an auxiliary variable is specified as a two element list, *VAR* is a variable initially bound to the result of evaluating the form *EXP*.

For example, given

(DEFMACRO FOO (A B &AUX C (D 5)) *FORM***)**

C will be bound to **NIL** and **D** to **5** when *FORM* is evaluated.

&WHOLE Used to get the whole macro calling form. Should be the first element of *ARGS*, and should be followed by a single litatom, which is set to the entire macro calling form. Other &-keywords or arguments can follow. For example, given

(DEFMACRO FOO (&WHOLE X A B) *FORM***)**

Expanding the macro call **(FOO 1 2)** would cause *FORM* to be evaluated with **X** set to **(FOO 1 2)**, **A** set to **1**, and **B** set to **2**.

DEFMACRO macros also support argument list "destructuring," a facility for accessing the structure of individual arguments to a macro. Any place in an argument list where a litatom is expected, an argument list (in the form described above) can appear instead. Such an embedded argument list is used to match the corresponding parts of that particular argument, which should be a list structure in the same form. In the simplest case, where the embedded argument list does not include &-keywords, this provides a simple way of picking apart list structures passed as arguments to a macro. For example, given

(DEFMACRO FOO (A (B (C . D)) E) *FORM***)**

Expanding the macro call **(FOO 1 (2 (3 4 5)) 6)** would cause *FORM* to be evaluated with with **A** set to **1**, **B** set to **2**, **C** set to **3**, **D** set to **(4 5)**, and **E** set to **6**. Note that the embedded argument list **(B (C . D))** has an embedded argument list **(C . D)**. Also notice that if an argument list ends in a dotted pair, that the final litatom matches the rest of the arguments in the macro call.

An embedded argument list can also include &-keywords, for interpreting parts of embedded list structures as if they appeared in a top-level macro call. For example, given

(DEFMACRO FOO (A (B &OPTIONAL (C 6)) D) *FORM***)**

Expanding the macro call **(FOO 1 (2) 3)** would cause *FORM* to be evaluated with with **A** set to **1**, **B** set to **2**, **C** set to **6** (because of the default value), and **D** set to **3**.

Warning: Embedded argument lists can only appear in positions in an argument list where a list is otherwise not accepted. In the above example, it would not be possible to specify an embedded argument list after the **&OPTIONAL** keyword, because it would be interpreted as an optional argument specification (with variable name, default value, set variable). However, it would be possible to specify an embedded argument list as the first element of an optional argument specification list, as so:

(DEFMACRO FOO (A (B &OPTIONAL ((X (Y) Z) '(1 (2) 3))) D) *FORM***)**

In this case, **X**, **Y**, and **Z** default to **1**, **2**, and **3**, respectively. Note that the "default" value has to be an appropriate list structure. Also, in this case either the whole structure **(X (Y) Z)** can be supplied, or it can be defaulted (i.e. is not possible to specify **X** while letting **Y** default).

10.6.2 Interpreting Macros

When the interpreter encounters a form **CAR** of which is an undefined function, it tries interpreting it as a macro. If **CAR** of the form has a macro definition, the macro is expanded, and the result of this expansion is evaluated in place of the original form. **CLISPTRAN** (page 21.25) is used to save the result of this expansion so that the expansion only has to be done once. On subsequent occasions, the translation (expansion) is retrieved from **CLISPARRAY** the same as for other CLISP constructs.

Note: Because of the way that the evaluator processes macros, if you have a macro on **FOO**, then typing **(FOO 'A 'B)** will work, but **FOO(A B)** will not work.

Sometimes, macros contain calls to functions that assume that the macro is being compiled. The variable **SHOULDCOMPILEMACROATOMS** is a list of functions that should be compiled to work correctly (initially **(OPCODES)** in Interlisp-D, **(ASSEMBLE LOC)** in Interlisp-10). **UNSAFEMACROATOMS** is a list of functions which effect the operation of the compiler, so such macro forms shouldn't even be expanded except by the compiler (initially **NIL** in Interlisp-D, **(C2EXP STORIN CEXP COMP)** in Interlisp-10). If the interpreter encounters a macro containing calls to functions on these two lists, instead of the macro being expanded, a dummy function is created with the form as its definition, and the dummy function is then compiled. A form consisting of a call to this dummy function with no arguments is then evaluated in place of the original form, and **CLISPTRAN** is used to save the translation as described above. There are some situations for which this procedure is not amenable, e.g. a **GO** inside the form which is being compiled will cause the compiler to give an **UNDEFINED TAG** error message because it is not compiling the entire function, just a part of it.

TABLE OF CONTENTS

[This page intentionally left blank]

11. VARIABLE BINDINGS AND THE INTERLISP STACK

A number of schemes have been used in different implementations of Lisp for storing the values of variables. These include:

(1) Storing values on an association list paired with the variable names.

(2) Storing values on the property list of the atom which is the name of the variable.

(3) Storing values in a special value cell associated with the atom name, putting old values on the function call stack, and restoring these values when exiting from a function.

(4) Storing values on on the function call stack.

Interlisp-10 uses the third scheme, so called "shallow binding". When a function is entered, the value of each variable bound by the function (function argument) is stored in a value cell associated with that variable name. The value that was in the value cell is stored in a block of storage called the basic frame for this function call. In addition, on exit from the function each variable must be individually unbound; that is, the old value saved in the basic frame must be restored to the value cell. Thus there is a higher cost for binding and unbinding a variable than in the fourth scheme, "deep binding". However, to find the current value of any variable, it is only necessary to access the variable's value cell, thus making variable reference considerably cheaper under shallow binding than under deep binding, especially for free variables. However, the shallow binding scheme used does require an additional overhead in switching contexts when doing "spaghetti stack" operations.

Interlisp-D uses the forth scheme, "deep binding." Every time a function is entered, a basic frame containing the new variables is put on top of the stack. Therefore, any variable reference requires searching the stack for the first instance of that variable, which makes free variable use somewhat more expensive than in a shallow binding scheme. On the other hand, spaghetti stack operations are considerably faster. Some other tricks involving copying freely-referenced variables to higher frames on the stack are also used to speed up the search.

The basic frames are allocated on a stack; for most user purposes, these frames should be thought of as containing the variable names associated with the function call, and the *current* values for that frame. The descriptions of the stack functions in below are presented from this viewpoint. Both interpreted and compiled functions store both the names and values of variables so that interpreted and compiled functions are compatible and can be freely intermixed, i.e., free variables can be used with no **SPECVAR** declarations necessary. However, it is possible to *suppress* storing of names in compiled functions, either for efficiency or to avoid a clash, via a **LOCALVAR** declaration (see page 18.5). The names are also very useful in debugging, for they make possible a complete symbolic backtrace in case of error.

In addition to the binding information, additional information is associated with each function call: access information indicating the path to search the basic frames for variable bindings, control information, and temporary results are also stored on the stack in a block called the frame extension. The interpreter also stores information about partially evaluated expressions as described on page 11.14.

11.1 The Spaghetti Stack

The Bobrow/Wegbreit paper, "A Model and Stack Implementation for Multiple Environments" (*Communications of the ACM,* Vol. 16, 10, October 1973.), describes an access and control mechanism more general than a simple linear stack. The access and control mechanism used by Interlisp is a slightly modified version of the one proposed by Bobrow and Wegbreit. This mechanism is called the "spaghetti stack."

The spaghetti system presents the access and control stack as a data structure composed of "frames." The functions described below operate on this structure. These primitives allow user functions to manipulate the stack in a machine independent way. Backtracking, coroutines, and more sophisticated control schemes can be easily implemented with these primitives.

The evaluation of a function requires the allocation of storage to hold the values of its local variables during the computation. In addition to variable bindings, an activation of a function requires a return link (indicating where control is to go after the completion of the computation) and room for temporaries needed during the computation. In the spaghetti system, one "stack" is used for storing all this information, but it is best to view this stack as a tree of linked objects called frame extensions (or simply frames).

A frame extension is a variable sized block of storage containing a frame name, a pointer to some variable bindings (the BLINK), and two pointers to other frame extensions (the ALINK and CLINK). In addition to these components, a frame extension contains other information (such as temporaries and reference counts) that does not interest us here.

The block of storage holding the variable bindings is called a basic frame. A basic frame is essentially an array of pairs, each of which contains a variable name and its value. The reason frame extensions point to basic frames (rather than just having them "built in") is so that two frame extensions can share a common basic frame. This allows two processes to communicate via shared variable bindings.

The chain of frame extensions which can be reached via the successive ALINKs from a given frame is called the "access chain" of the frame. The first frame in the access chain is the starting frame. The chain through successive CLINKs is called the "control chain".

A frame extension completely specifies the variable bindings and control information necessary for the evaluation of a function. Whenever a function (or in fact, any form which generally binds local variables) is evaluated, it is associated with some frame extension.

In the beginning there is precisely one frame extension in existence. This is the frame in which the top-level call to the interpreter is being run. This frame is called the "top-level" frame.

Since precisely one function is being executed at any instant, exactly one frame is distinguished as having the "control bubble" in it. This frame is called the active frame. Initially, the top-level frame is the active frame. If the computation in the active frame invokes another function, a new basic frame and frame extension are built. The frame name of this basic frame will be the name of the function being called. The ALINK, BLINK, and CLINK of the new frame all depend on precisely how the function is invoked. The new function is then run in this new frame by passing control to that frame, i.e., it is made the active frame.

Once the active computation has been completed, control normally returns to the frame pointed to by the CLINK of the active frame. That is, the frame in the CLINK becomes the active frame.

In most cases, the storage associated with the basic frame and frame extension just abandoned can be reclaimed. However, it is possible to obtain a pointer to a frame extension and to "hold on" to this frame even after it has been exited. This pointer can be used later to run another computation in that environment, or even "continue" the exited computation.

A separate data type, called a stack pointer, is used for this purpose. A stack pointer is just a cell that literally points to a frame extension. Stack pointers print as *#ADR/FRAMENAME*, e.g., **#1,13636/COND**. Stack pointers are returned by many of the stack manipulating functions described below. Except for certain abbreviations (such as "the frame with such-and-such a name"), stack pointers are the only way the user can reference a frame extension. As long as the user has a stack pointer which references a frame extension, that frame extension (and all those that can be reached from it) will not be garbage collected.

Note that two stack pointers referencing the same frame extension are *not* necessarily **EQ**, i.e., **(EQ (STKPOS 'FOO) (STKPOS 'FOO)) = NIL**. However, **EQP** can be used to test if two different stack pointers reference the same frame extension (page 9.3).

It is possible to evaluate a form with respect to an access chain other than the current one by using a stack pointer to refer to the head of the access chain desired. Note, however, that this can be very expensive when using a shallow binding scheme such as that in Interlisp-10. When evaluating the form, since all references to variables under the shallow binding scheme go through the variable's value cell, the values in the value cells must be adjusted to reflect the values appropriate to the desired access chain. This is done by changing all the bindings on the current access chain (all the name-value pairs) so that they contain the value current at the time of the call. Then along the new access path, all bindings are made to contain the previous value of the variable, and the current value is placed in the value cell. For that part of the access path which is shared by the old and new chain, no work has to be done. The context switching time, i.e. the overhead in switching from the current, active, access chain to another one, is directly proportional to the size of the two branches that are not shared between the access contexts. This cost should be remembered in using generators and coroutines (page 11.16).

11.2 Stack Functions

In the descriptions of the stack functions below, when we refer to an argument as a stack descriptor, we mean that it is one of the following:

A stack pointer — A stack pointer is an object that points to a frame on the stack. Stack pointers are returned by many of the stack manipulating functions described below.

NIL — Specifies the active frame; that is, the frame of the stack function itself.

T — Specifies the top-level frame.

A litatom — Specifies the first frame (along the control chain from the active frame) that has the frame name *LITATOM*. Equivalent to **(STKPOS** *LITATOM* **-1)**.

A list of litatoms — Specifies the first frame (along the control chain from the active frame) whose frame name is included in the list.

A number *N* — Specifies the *N*th frame back from the active frame. If *N* is negative, the control chain is followed, otherwise the access chain is followed. Equivalent to **(STKNTH** *N***)**

In the stack functions described below, the following errors can occur: The error **ILLEGAL STACK ARG** occurs when a stack descriptor is expected and the supplied argument is either not a legal stack descriptor (i.e., not a stack pointer, litatom, or number), or is a litatom or number for which there is no corresponding stack frame, e.g., **(STKNTH -1 'FOO)** where there is no frame named **FOO** in the active control chain or **(STKNTH -10 'EVALQT)**. The error **STACK POINTER HAS BEEN RELEASED** occurs whenever a released stack pointer is supplied as a stack descriptor argument for any purpose other than as a stack pointer to re-use.

Note: The creation of a single stack pointer can result in the retention of a large amount of stack space. Therefore, one should try to release stack pointers when they are no longer needed (see page 11.9).

11.2.1 Searching the Stack

(STKPOS *FRAMENAME N POS OLDPOS***)** [Function]

Returns a stack pointer to the *N*th frame with frame name *FRAMENAME*. The search begins with (and includes) the frame specified by the stack descriptor *POS*. The search proceeds along the control chain from *POS* if *N* is negative, or along the access chain if *N* is positive. If *N* is **NIL**, **-1** is used. Returns a stack pointer to the frame if such a frame exists, otherwise returns **NIL**. If *OLDPOS* is supplied and is a stack pointer, it is reused. If *OLDPOS* is supplied and is a stack pointer and **STKPOS** returns **NIL**, *OLDPOS* is released. If *OLDPOS* is not a stack pointer it is ignored.

Note: **(STKPOS 'STKPOS)** causes an error, **ILLEGAL STACK ARG**; it is not permissible to create a stack pointer to the active frame.

(STKNTH ***N POS OLDPOS*****)** [Function]

Returns a stack pointer to the *N*th frame back from the frame specified by the stack descriptor *POS*. If *N* is negative, the control chain from *POS* is followed. If *N* is positive the access chain is followed. If *N* equals 0, **STKNTH** returns a stack pointer to *POS* (this provides a way to copy a stack pointer). Returns **NIL** if there are fewer than *N* frames in the appropriate chain. If *OLDPOS* is supplied and is a stack pointer, it is reused. If *OLDPOS* is not a stack pointer it is ignored.

Note: **(STKNTH 0)** causes an error, **ILLEGAL STACK ARG**; it is not possible to create a stack pointer to the active frame.

(STKNAME ***POS*****)** [Function]

Returns the frame name of the frame specified by the stack descriptor *POS*.

(SETSTKNAME ***POS NAME*****)** [Function]

Changes the frame name of the frame specified by *POS* to be *NAME*. Returns *NAME*.

(STKNTHNAME ***N POS*****)** [Function]

Returns the frame name of the *N*th frame back from *POS*. Equivalent to **(STKNAME (STKNTH** ***N POS*****))** but avoids creation of a stack pointer.

In summary, **STKPOS** converts function names to stack pointers, **STKNTH** converts numbers to stack pointers, **STKNAME** converts stack pointers to function names, and **STKNTHNAME** converts numbers to function names.

11.2.2 Variable Bindings in Stack Frames

The following functions are used for accessing and changing bindings. Some of functions take an argument, *N*, which specifies a particular binding in the basic frame. If *N* is a literal atom, it is assumed to be the name of a variable bound in the basic frame. If *N* is a number, it is assumed to reference the *N*th binding in the basic frame. The first binding is 1. If the basic frame contains no binding with the given name or if the number is too large or too small, the error **ILLEGAL ARG** occurs.

(STKSCAN ***VAR IPOS OPOS*****)** [Function]

Searches beginning at *IPOS* for a frame in which a variable named *VAR* is bound. The search follows the access chain. Returns a stack pointer to the frame if found, otherwise returns **NIL**. If *OPOS* is a stack pointer it is reused, otherwise it is ignored.

(FRAMESCAN *ATOM POS***)** [Function]

Returns the relative position of the binding of *ATOM* in the basic frame of *POS*. Returns **NIL** if *ATOM* is not found.

(STKARG *N POS —***)** [Function]

Returns the value of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or number.

(STKARGNAME *N POS***)** [Function]

Returns the name of the binding specified by *N*, in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or number.

(SETSTKARG *N POS VAL***)** [Function]

Sets the value of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns *VAL*.

(SETSTKARGNAME *N POS NAME***)** [Function]

Sets the variable name to *NAME* of the binding specified by *N* in the basic frame of the frame specified by the stack descriptor *POS*. *N* can be a literal atom or a number. Returns *NAME*.

(STKNARGS *POS —***)** [Function]

Returns the number of arguments bound in the basic frame of the frame specified by the stack descriptor *POS*.

(VARIABLES *POS***)** [Function]

Returns a list of the variables bound at *POS*.

(STKARGS *POS —***)** [Function]

Returns a list of the values of the variables bound at *POS*.

11.2.3 Evaluating Expressions in Stack Frames

The following functions are used to evaluate an expression in a different environment:

(ENVEVAL *FORM APOS CPOS AFLG CFLG***)** [Function]

Evaluates *FORM* in the environment specified by *APOS* and *CPOS*. That is, a new active frame is created with the frame specified by the stack descriptor *APOS* as its ALINK, and the frame specified by the stack descriptor *CPOS* as its CLINK. Then *FORM* is evaluated. If *AFLG* is not **NIL**, and *APOS* is a stack pointer, then

APOS will be released. Similarly, if *CFLG* is not **NIL**, and *CPOS* is a stack pointer, then *CPOS* will be released.

(ENVAPPLY *FN ARGS APOS CPOS AFLG CFLG***)** [Function]

APPLYs *FN* to *ARGS* in the environment specified by *APOS* and *CPOS*. *AFLG* and *CFLG* have the same interpretation as with **ENVEVAL**.

(EVALV *VAR POS RELFLG***)** [Function]

Evaluates *VAR*, where *VAR* is assumed to be a litatom, in the access environment specifed by the stack descriptor *POS*. If *VAR* is unbound, **EVALV** returns **NOBIND** and does not generate an error. If *RELFLG* is non-**NIL** and *POS* is a stack pointer, it will be released after the variable is looked up. While **EVALV** could be defined as **(ENVEVAL** *VAR POS* **NIL** *RELFLG***)** it is in fact somewhat faster.

(STKEVAL *POS FORM FLG* **—)** [Function]

Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*. If *FLG* is not **NIL** and *POS* is a stack pointer, releases *POS*. The definition of **STKEVAL** is **(ENVEVAL** *FORM POS* **NIL** *FLG***)**.

(STKAPPLY *POS FN ARGS FLG***)** [Function]

Similar to **STKEVAL** but applies *FN* to *ARGS*.

11.2.4 Altering Flow of Control

The following functions are used to alter the normal flow of control, possibly jumping to a different frame on the stack. **RETEVAL** and **RETAPPLY** allow evaluating an expression in the specified environment first.

(RETFROM *POS VAL FLG***)** [Function]

Return from the frame specified by the stack descriptor *POS*, with the value *VAL*. If *FLG* is not **NIL**, and *POS* is a stack pointer, then *POS* is released. An attempt to **RETFROM** the top level (e.g., **(RETFROM T))** causes an error, **ILLEGAL STACK ARG**. **RETFROM** can be written in terms of **ENVEVAL** as follows:

```
(RETFROM
 (LAMBDA (POS VAL FLG)
  (ENVEVAL (LIST 'QUOTE VAL)
      NIL
      (if (STKNTH -1 POS (if FLG then POS))
        else (ERRORX (LIST 19 POS)))
```

```
NIL
T)))
```

(RETTO *POS VAL FLG***)** [Function]

Like **RETFROM**, except returns *to* the frame specified by *POS*.

(RETEVAL *POS FORM FLG —***)** [Function]

Evaluates *FORM* in the access environment of the frame specified by the stack descriptor *POS*, and then returns from *POS* with that value. If *FLG* is not **NIL** and *POS* is a stack pointer, then *POS* is released. The definition of **RETEVAL** is equivalent to **(ENVEVAL** *FORM POS* **(STKNTH -1** *POS***)** *FLG* **T)**, except that **RETEVAL** does not create a stack pointer.

(RETAPPLY *POS FN ARGS FLG***)** [Function]

Similar to **RETEVAL** except applies *FN* to *ARGS*.

11.2.5 Releasing and Reusing Stack Pointers

The following functions and variables are used for manipulating stack pointers:

(STACKP *X***)** [Function]

Returns *X* if *X* is a stack pointer, otherwise returns **NIL**.

(RELSTK *POS***)** [Function]

Release the stack pointer *POS* (see below). If *POS* is not a stack pointer, does nothing. Returns *POS*.

(RELSTKP *X***)** [Function]

Returns **T** is *X* is a released stack pointer, **NIL** otherwise.

(CLEARSTK *FLG***)** [Function]

If *FLG* is **NIL**, releases all active stack pointers, and returns **NIL**. If *FLG* is **T**, returns a list of all the active (unreleased) stack pointers.

CLEARSTKLST [Variable]

A variable used by the top-level executive. Every time the top-level executive is re-entered (e.g., following errors, or control-D), **CLEARSTKLST** is checked. If its value is **T**, all active stack pointers are released using **CLEARSTK**. If its value is a list, then all stack pointers on that list are released. If its value is **NIL**, nothing is released. **CLEARSTKLST** is initially **T**.

NOCLEARSTKLST [Variable]

A variable used by the top-level executive. If **CLEARSTKLST** is **T** (see above) all active stack pointers *except* those on **NOCLEARSTKLST** are released. **NOCLEARSTKLST** is initially **NIL**.

Note: If one wishes to use multiple environments that survive through control-D, either **CLEARSTKLST** should be set to **NIL**, or else those stack pointers to be retained should be explicitly added to **NOCLEARSTKLST**.

The creation of a single stack pointer can result in the retention of a large amount of stack space. Furthermore, this space will not be freed until the next garbage collection, *even if the stack pointer is no longer being used*, unless the stack pointer is explicitly released or reused. If there is sufficient amount of stack space tied up in this fashion, a **STACK OVERFLOW** condition can occur, even in the simplest of computations. For this reason, the user should consider releasing a stack pointer when the environment referenced by the stack pointer is no longer needed.

The effects of releasing a stack pointer are:

(1) The link between the stack pointer and the stack is broken by setting the contents of the stack pointer to the "released mark" (currently unboxed 0). A released stack pointer prints as *#ADRI#0*.

(2) If this stack pointer was the last remaining reference to a frame extension; that is, if no other stack pointer references the frame extension and the extension is not contained in the active control or access chain, then the extension may be reclaimed, and is reclaimed immediately. The process repeats for the access and control chains of the reclaimed extension so that all stack space that was reachable only from the released stack pointer is reclaimed.

A stack pointer may be released using the function **RELSTK**, but there are some cases for which **RELSTK** is not sufficient. For example, if a function contains a call to **RETFROM** in which a stack pointer was used to specify where to return to, it would not be possible to simultaneously release the stack pointer. (A **RELSTK** appearing in the function following the call to **RETFROM** would not be executed!) To permit release of a stack pointer in this situation, the stack functions that relinquish control have optional flag arguments to denote whether or not a stack pointer is to be released (*AFLG* and *CFLG*). Note that in this case releasing the stack pointer will *not* cause the stack space to be reclaimed immediately because the frame referenced by the stack pointer will have become part of the active environment.

Another way of avoiding creating new stack pointers is to *reuse* stack pointers that are no longer needed. The stack functions that create stack pointers (**STKPOS, STKNTH**, and **STKSCAN**) have an optional argument which is a stack pointer to reuse. When a stack pointer is reused, two things happen. First the stack pointer is released (see above). Then the pointer to the new frame extension is deposited in the stack pointer. The old stack pointer (with its new contents) is the value of the function. Note that the reused stack pointer will be released even if the function does not find the specified frame.

Note that even if stack pointers are explicitly being released, *creation* of many stack pointers can cause a garbage collection of stack pointer space. Thus, if the user's application requires creating many stack pointers, he definitely should take advantage of reusing stack pointers.

11.2.6 Backtrace Functions

The following functions perform a "backtrace," printing information about every frame on the stack. Arguments allow only backtracing a selected range of the stack, skipping selected frames, and printing different amounts of information about each frame.

(BACKTRACE *IPOS EPOS FLAGS FILE PRINTFN***)** [Function]

Performs a backtrace beginning at the frame specified by the stack descriptor *IPOS*, and ending with the frame specified by the stack descriptor *EPOS*. *FLAGS* is a number in which the options of the **BACKTRACE** are encoded. If a bit is set, the corresponding information is included in the backtrace.

bit 0 - print arguments of non-**SUBR**s.

bit 1 - print temporaries of the interpreter.

bit 2 - print **SUBR** arguments and local variables.

bit 3 - omit printing of **UNTRACE:** and function names.

bit 4 - follow access chain instead of control chain.

bit 5 - print temporaries, i.e. the blips (see page 11.14).

For example: If *FLAGS* = **47Q**, everything is printed. If *FLAGS* = **21Q**, follows the access chain, prints arguments.

FILE is the file that the backtrace is printed to. *FILE* must be open. *PRINTFN* is used when printing the values of variables, temporaries, blips, etc. *PRINTFN* = **NIL** defaults to **PRINT**.

(BAKTRACE *IPOS EPOS SKIPFNS FLAGS FILE***)** [Function]

Prints a backtrace from *IPOS* to *EPOS* onto *FILE*. *FLAGS* specifies the options of the backtrace, e.g., do/don't print arguments, do/don't print temporaries of the interpreter, etc., and is the same as for **BACKTRACE**.

SKIPFNS is a list of functions. As **BAKTRACE** scans down the stack, the stack name of each frame is passed to each function in *SKIPFNS*, and if any of them return non-**NIL**, *POS* is skipped (including all variables).

BAKTRACE collapses the sequence of several function calls corresponding to a call to a system package into a single "function" using **BAKTRACELST** as described below. For example, any call to the editor is printed as ****EDITOR****, a break is printed as ****BREAK****, etc.

BAKTRACE is used by the **BT**, **BTV**, **BTV+**, **BTV***, and **BTV!** break commands, with *FLAGS* = **0**, **1**, **5**, **7**, and **47Q** respectively.

Note: **BAKTRACE** calls **BACKTRACE** with a *PRINTFN* of **SHOWPRINT** (page 25.10), so that if **SYSPRETTYFLG = T**, the values will be prettyprinted.

BAKTRACELST [Variable]

Used for telling **BAKTRACE** (therefore, the **BT**, **BTV**, etc. commands) to abbreviate various sequences of function calls on the stack by a single key, e.g. ****BREAK****, ****EDITOR****, etc.

The operation of **BAKTRACE** and format of **BAKTRACELST** is described so that the user can add his own entries to **BAKTRACELST**. Each entry on **BAKTRACELST** is a list of the form **(***FRAMENAME KEY* . *PATTERN***)** or **(***FRAMENAME* **(***KEY_1* . *$PATTERN_1$***)** ... **(***KEY_N* . *$PATTERN_N$***))**, where a pattern is a list of elements that are either atoms, which match a single frame, or lists, which are interpreted as a list of alternative patterns, e.g. **(PROGN **BREAK** EVAL ((ERRORSET BREAK1A BREAK1) (BREAK1)))**

BAKTRACE operates by scanning up the stack and, at each point, comparing the current frame name, with the frame names on **BAKTRACELST**, i.e. it does an **ASSOC**. If the frame name does appear, **BAKTRACE** attempts to match the stack as of that point with (one of) the patterns. If the match is successful, **BAKTRACE** prints the corresponding key, and continues with where the match left off. If the frame name does not appear, or the match fails, **BAKTRACE** simply prints the frame name and continues with the next higher frame (unless the *SKIPFNS* applied to the frame name are non-**NIL** as described above).

Matching is performed by comparing atoms in the pattern with the current frame name, and matching lists as patterns, i.e. sequences of function calls, always working up the stack. For example, either of the sequence of function calls "... **BREAK1 BREAK1A ERRORSET EVAL PROGN** ..." or "... **BREAK1 EVAL**

PROGN ..." would match with the sample entry given above, causing ****BREAK**** to be printed.

Special features:

- The litatom **&** can be used to match any frame.
- The pattern "**-**" can be used to match nothing. **-** is useful for specifying an optional match, e.g. the example above could also have been written as **(PROGN **BREAK** EVAL ((ERRORSET BREAK1A) -) BREAK1)**.
- It is not necessary to provide in the pattern for matching dummy frames, i.e. frames for which **DUMMYFRAMEP** (see page 11.13) is true, e.g. in Interlisp-10, ***PROG*LAM**, ***ENV***, **NOLINKDEF1**, etc. When working on a match, the matcher automatically skips over these frames when they do not match.
- If a match succeeds and the *KEY* is **NIL**, nothing is printed. For example, **(*PROG*LAM NIL EVALA *ENV)**. This sequence will occur following an error which then causes a break if some of the function's arguments are **LOCALVARS**.

11.2.7 Other Stack Functions

(DUMMYFRAMEP *POS***)** [Function]

Returns **T** if the user never wrote a call to the function at *POS*, e.g. in Interlisp-10, **DUMMYFRAMEP** is **T** for ***PROG*LAM**, ***ENV***, and **FOOBLOCK** frames (see block compiler, page 18.17).

REALFRAMEP and **REALSTKNTH** can be used to write functions which manipulate the stack and work on either interpreted or compiled code:

(REALFRAMEP *POS INTERPFLG***)** [Function]

Returns *POS* if *POS* is a "real" frame, i.e. if *POS* is not a dummy frame and *POS* is a frame that does not disappear when compiled (such as **COND**); otherwise **NIL**. If *INTERPFLG* = **T**, returns *POS* if *POS* is not a dummy frame. For example, if **(STKNAME** *POS***) = COND**, **(REALFRAMEP** *POS***)** is **NIL**, but **(REALFRAMEP** *POS* **T)** is *POS*.

(REALSTKNTH *N POS INTERPFLG OLDPOS***)** [Function]

Returns a stack pointer to the *N*th (or -*N*th) frames for which **(REALFRAMEP** *POS INTERPFLG***)** is *POS*.

(MAPDL *MAPDLFN MAPDLPOS***)** [Function]

Starts at *MAPDLPOS* and applies the function *MAPDLFN* to two arguments (the frame name and a stack pointer to the frame), for each frame until the top of the stack is reached. Returns **NIL**. For example,

```
[MAPDL (FUNCTION (LAMBDA (X POS)
        (if (IGREATERP (STKNARGS POS) 2)
          then (PRINT X)]
```

will print all functions of more than two arguments.

(SEARCHPDL *SRCHFN SRCHPOS***)** [Function]

Similar to **MAPDL**, except searches the stack starting at position *SRCHPOS* until it finds a frame for which *SRCHFN*, a function of two arguments applied to the *name* of the frame and the frame itself, is not **NIL**. Returns **(***NAME* . *FRAME***)** if such a frame is found, otherwise **NIL**.

11.3 The Stack and the Interpreter

In addition to the names and values of arguments for functions, information regarding partially-evaluated expressions is kept on the push-down list. For example, consider the following definition of the function **FACT** (intentionally faulty):

```
(FACT
  [LAMBDA (N)
    (COND
      ((ZEROP N)
       L)
      (T (ITIMES N (FACT (SUB1 N])
```

In evaluating the form **(FACT 1)**, as soon as **FACT** is entered, the interpreter begins evaluating the implicit **PROGN** following the **LAMBDA**. The first function entered in this process is **COND**. **COND** begins to process its list of clauses. After calling **ZEROP** and getting a **NIL** value, **COND** proceeds to the next clause and evaluates **T**. Since **T** is true, the evaluation of the implicit **PROGN** that is the consequent of the **T** clause is begun. This requires calling the function **ITIMES**. However before **ITIMES** can be called, its arguments must be evaluated. The first argument is evaluated by retrieving the current binding of **N** from its value cell; the second involves a recursive call to **FACT**, and another implicit **PROGN**, etc.

Note that at each stage of this process, some portion of an expression has been evaluated, and another is awaiting evaluation. The output below (from Interlisp-10) illustrates this by showing the state of the push-down list at the point in the computation of **(FACT 1)** when the unbound atom **L** is reached.

```
←FACT(1)
```

```
u.b.a. L {in FACT} in ((ZEROP N) L)
(L broken)
:BTV!

  *TAIL* (L)

  *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
COND

  *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
  *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))

  N 0
FACT

  *FORM* (FACT (SUB1 N))
  *FN* ITIMES
  *TAIL* ((FACT (SUB1 N)))
  *ARGVAL* 1
  *FORM* (ITIMES N (FACT (SUB1 N)))
  *TAIL* ((ITIMES N (FACT (SUB1 N))))

  *ARG1 (((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
COND

  *FORM* (COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N)))))
  *TAIL* ((COND ((ZEROP N) L) (T (ITIMES N (FACT (SUB1 N))))))

  N 1
FACT

**TOP**
```

Internal calls to **EVAL**, e.g., from **COND** and the interpreter, are marked on the push-down list by a special mark or blip which the backtrace prints as ***FORM***. The genealogy of ***FORM***'s is thus a history of the computation. Other temporary information stored on the stack by the interpreter includes the tail of a partially evaluated implicit **PROGN** (e.g., a cond clause or lambda expression) and the tail of a partially evaluated form (i.e., those arguments not yet evaluated), both indicated on the backtrace by ***TAIL***, the values of arguments that have already been evaluated, indicated by ***ARGVAL***, and the names of functions waiting to be called, indicated by ***FN***. ***ARG1**, ..., ***ARGn** are used by the backtrace to indicate the (unnamed) arguments to **SUBR**s.

Note that a function is not actually entered and does not appear on the stack, until its arguments have been evaluated (except for nlambda functions, of course). Also note that the ***ARG1**,

FORM, ***TAIL***, etc. "bindings" comprise the actual working storage. In other words, in the above example, if a (lower) function changed the value of the ***ARG1** binding, the **COND** would continue interpreting the new binding as a list of **COND** clauses. Similarly, if the ***ARGVAL*** binding were changed, the new value would be given to **ITIMES** as its first argument after its second argument had been evaluated, and **ITIMES** was actually called.

Note that ***FORM***, ***TAIL***, ***ARGVAL***, etc., do not actually appear as variables on the stack, i.e., evaluating ***FORM*** or calling **STKSCAN** to search for it will not work. However, the functions **BLIPVAL**, **SETBLIPVAL**, and **BLIPSCAN** described below are available for accessing these internal blips. These functions currently know about four different types of blips:

FN The name of a function about to be called.

ARGVAL An argument for a function about to be called.

FORM A form in the process of evaluation.

TAIL The tail of a **COND** clause, implicit **PROGN**, **PROG**, etc.

(BLIPVAL *BLIPTYP IPOS FLG***)** [Function]

Returns the value of the specified blip of type *BLIPTYP*. If *FLG* is a number **N**, finds the **N**th blip of the desired type, searching the control chain beginning at the frame specified by the stack descriptor *IPOS*. If *FLG* is **NIL**, 1 is used. If *FLG* is **T**, returns the number of blips of the specified type at *IPOS*.

(SETBLIPVAL *BLIPTYP IPOS N VAL***)** [Function]

Sets the value of the specified blip of type *BLIPTYP*. Searches for the *N*th blip of the desired type, beginning with the frame specified by the stack descriptor *IPOS*, and following the control chain.

(BLIPSCAN *BLIPTYP IPOS***)** [Function]

Returns a stack pointer to the frame in which a blip of type *BLIPTYP* is located. Search begins at the frame specified by the stack descriptor *IPOS* and follows the control chain.

11.4 Generators

A *generator* is like a subroutine except that it retains information about previous times it has been called. Some of this state may be data (for example, the seed in a random number generator), and some may be in program state (as in a recursive generator which finds all the atoms in a list structure). For example, if **LISTGEN** is defined by:

```
(DEFINEQ (LISTGEN (L)
   (if L then (PRODUCE (CAR L))
         (LISTGEN (CDR L))))
```

we can use the function **GENERATOR** (described below) to create a generator that uses **LISTGEN** to produce the elements of a list one at a time, e.g.,

(SETQ GR (GENERATOR (LISTGEN '(A B C))))

creates a generator, which can be called by

(GENERATE GR)

to produce as values on successive calls, **A**, **B**, **C**. When **GENERATE** (not **GENERATOR**) is called the first time, it simply starts evaluating **(LISTGEN '(A B C))**. **PRODUCE** gets called from **LISTGEN**, and pops back up to **GENERATE** with the indicated value after saving the state. When **GENERATE** gets called again, it continues from where the last **PRODUCE** left off. This process continues until finally **LISTGEN** completes and returns a value (it doesn't matter what it is). **GENERATE** then returns **GR** itself as its value, so that the program that called **GENERATE** can tell that it is finished, i.e., there are no more values to be generated.

(GENERATOR *FORM COMVAR***)** [NLambda Function]

An nlambda function that creates a generator which uses *FORM* to compute values. **GENERATOR** returns a *generator handle* which is represented by a dotted pair of stack pointers.

COMVAR is optional. If its value (**EVAL** of) is a generator handle, the list structure and stack pointers will be reused. Otherwise, a new generator handle will be constructed.

GENERATOR compiles open.

(PRODUCE *VAL***)** [Function]

Used from within a generator to return *VAL* as the value of the corresponding call to **GENERATE**.

(GENERATE *HANDLE VAL***)** [Function]

Restarts the generator represented by *HANDLE*. *VAL* is returned as the value of the **PRODUCE** which last suspended the operation of the generator. When the generator runs out of values, **GENERATE** returns *HANDLE* itself.

Examples:

The following function will go down recursively through a list structure and produce the atoms in the list structure one at a time.

```
(DEFINEQ (LEAVESG (L)
  (if (ATOM L)
   then (PRODUCE L)
   else (LEAVESG (CAR L))
     (if (CDR L)
       then (LEAVESG (CDR L)]
```

The following function prints each of these atoms as it appears. It illustrates how a loop can be set up to use a generator.

```
(DEFINEQ (PLEAVESG1 (L)
  (PROG (X LHANDLE)
     (SETQ LHANDLE (GENERATOR (LEAVESG L)))
   LP (SETQ X (GENERATE LHANDLE))
     (if (EQ X LHANDLE)
       then (RETURN NIL))
     (PRINT X)
     (GO LP))]
```

Note that the loop terminates when the value of the generator is **EQ** to the dotted pair which is the value produced by the call to **GENERATOR**. A CLISP iterative operator, **OUTOF**, is provided which makes it much easier to write the loop in **PLEAVESG1**. **OUTOF** (or **outof**) can precede a form which is to be used as a generator. On each iteration, the iteration variable will be set to successive values returned by the generator; the loop will be terminated automatically when the generator runs out. Therefore, the following is equivalent to the above program **PLEAVESG1**:

```
(DEFINEQ (PLEAVESG2 (L)
  (for X outof (LEAVESG L) do (PRINT x))]
```

Here is another example; the following form will print the first **N** atoms.

```
(for X outof (MAPATOMS (FUNCTION PRODUCE))
  as I from 1 to N do (PRINT X))
```

11.5 Coroutines

This package provides facilities for the creation and use of fully general coroutine structures. It uses a stack pointer to preserve the state of a coroutine, and allows arbitrary switching between *N* different coroutines, rather than just a call to a generator and return. This package is slightly more efficient than the generator package described above, and allows more flexibility on specification of what to do when a coroutine terminates.

(COROUTINE *CALLPTR COROUTPTR COROUTFORM ENDFORM***)** [NLambda Function]

This nlambda function is used to create a coroutine and initialize the linkage. *CALLPTR* and *COROUTPTR* are the names of two variables, which will be set to appropriate stack pointers. If the values of *CALLPTR* or *COROUTPTR* are already stack pointers, the stack pointers will be reused. *COROUTFORM* is the form which is evaluated to start the coroutine; *ENDFORM* is a form to be evaluated if *COROUTFORM* actually returns when it runs out of values.

COROUTINE compiles open.

(RESUME *FROMPTR TOPTR VAL***)** [Function]

Used to transfer control from one coroutine to another. *FROMPTR* should be the stack pointer for the current coroutine, which will be smashed to preserve the current state. *TOPTR* should be the stack pointer which has preserved the state of the coroutine to be transferred to, and *VAL* is the value that is to be returned to the latter coroutine as the value of the **RESUME** which suspended the operation of that coroutine.

For example, the following is the way one might write the **LEAVES** program using the coroutine package:

```
(DEFINEQ (LEAVESC (L COROUTPTR CALLPTR)
  (if (ATOM L)
   then (RESUME COROUTPTR CALLPTR L)
   else (LEAVESC (CAR L) COROUTPTR CALLPTR)
      (if (CDR L) then (LEAVESC (CDR L) COROUTPTR CALLPTR))))]
```

A function **PLEAVESC** which uses **LEAVESC** can be defined as follows:

```
(DEFINEQ (PLEAVESC (L)
  (bind PLHANDLE LHANDLE
  first (COROUTINE PLHANDLE LHANDLE
           (LEAVESC L LHANDLE PLHANDLE)
           (RETFROM 'PLEAVESC))
  do (PRINT (RESUME PLHANDLE LHANDLE))))]
```

By **RESUME**ing **LEAVESC** repeatedly, this function will print all the leaves of list **L** and then return out of **PLEAVESC** via the **RETFROM**. The **RETFROM** is necessary to break out of the non-terminating do-loop. This was done to illustrate the additional flexibility allowed through the use of *ENDFORM*.

We use two coroutines working on two trees in the example **EQLEAVES**, defined below. **EQLEAVES** tests to see whether two trees have the same leaf set in the same order, e.g., **(EQLEAVES '(A B C) '(A B (C)))** is true.

```
(DEFINEQ (EQLEAVES (L1 L2)
  (bind LHANDLE1 LHANDLE2 PE EL1 EL2
   first (COROUTINE PE LHANDLE1 (LEAVESC L1 LHANDLE1 PE)
'NO-MORE)
       (COROUTINE PE LHANDLE2 (LEAVESC L2 LHANDLE2 PE)
'NO-MORE)
   do (SETQ EL1 (RESUME PE LHANDLE1))
     (SETQ EL2 (RESUME PE LHANDLE2))
     (if (NEQ EL1 EL2)
      then (RETURN NIL))
   repeatuntil (EQ EL1 'NO-MORE)
   finally (RETURN T)))]
```

11.6 Possibilities Lists

A possibilities list is the interface between a generator and a consumer. The possibilities list is initialized by a call to **POSSIBILITIES**, and elements are obtained from it by using **TRYNEXT**. By using the spaghetti stack to maintain separate environments, this package allows a regime in which a generator can put a few items in a possibilities list, suspend itself until they have been consumed, and be subsequently aroused and generate some more.

(POSSIBILITIES *FORM*) [NLambda Function]

This nlambda function is used for the initial creation of a possibilities list. *FORM* will be evaluated to create the list. It should use the functions **NOTE** and **AU-REVOIR** described below to generate possibilities. Normally, one would set some variable to the possibilities list which is returned, so it can be used later, e.g.:

(SETQ PLIST (POSSIBILITIES (GENERFN V1 V2))).

POSSIBILITIES compiles open.

(NOTE *VAL LSTFLG*) [Function]

Used within a generator to put items on the possibilities list being generated. If *LSTFLG* is equal to **NIL**, *VAL* is treated as a single item. If *LSTFLG* is non-**NIL**, then the list *VAL* is **NCONC**ed on the end of the possibilities list. Note that it is perfectly reasonable to create a possibilities list using a second generator, and **NOTE** that list as possibilities for the current generator with

LSTFLG equal to **T**. The lower generator will be resumed at the appropriate point.

(AU-REVOIR *VAL***)** [NoSpread Function]

Puts *VAL* on the possibilities list if it is given, and then suspends the generator and returns to the consumer in such a fashion that control will return to the generator at the **AU-REVOIR** if the consumer exhausts the possibilities list.

Note: **NIL** is not put on the possibilities list unless it is explicitly given as an argument to **AU-REVOIR**, i.e., **(AU-REVOIR)** and **(AU-REVOIR NIL)** are *not* the same. **AU-REVOIR** and **ADIEU** are lambda nospreads to enable them to distinguish these two cases.

(ADIEU *VAL***)** [NoSpread Function]

Like **AU-REVOIR** except releases the generator instead of suspending it.

(TRYNEXT *PLST ENDFORM VAL***)** [NLambda Function]

This nlambda function allows a consumer to use a possibilities list. It removes the first item from the possibilities list named by *PLST* (i.e. *PLST* must be an atom whose value is a possiblities list), and returns that item, provided it is not a generator handle. If a generator handle is encountered, the generator is reawakened. When it returns a possibilities list, this list is added to the front of the current list. When a call to **TRYNEXT** causes a generator to be awakened, *VAL* is returned as the value of the **AU-REVOIR** which put that generator to sleep. If *PLST* is empty, it evaluates *ENDFORM* in the caller's environment.

TRYNEXT compiles open.

(CLEANPOSLST *PLST***)** [Function]

This function is provided to release any stack pointers which may be left in the *PLST* which was not used to exhaustion.

For example, **FIB** is a generator for fibonnaci numbers. It starts out by **NOTE**ing its two arguments, then suspends itself. Thereafter, on being re-awakened, it will **NOTE** two more terms in the series and suspends again. **PRINTFIB** uses **FIB** to print the first **N** fibonacci numbers.

```
(DEFINEQ (FIB (F1 F2)
 (do (NOTE F1)
   (NOTE F2)
   (SETQ F1 (IPLUS F1 F2))
   (SETQ F2 (IPLUS F1 F2))
   (AU-REVOIR)]
```

Note that this **AU-REVOIR** just suspends the generator and adds nothing to the possibilities list except the generator.

```
(DEFINEQ (PRINTFIB (N)
  (PROG ((FL (POSSIBILITIES (FIB 0 1))))
     (RPTQ N (PRINT (TRYNEXT FL)))
     (CLEANPOSLST FL)]
```

Note that **FIB** itself will never terminate.

[This page intentionally left blank]

12. MISCELLANEOUS

12.1 Greeting and Initialization Files

Many of the features of Interlisp are controlled by variables that the user can adjust to his or her own tastes. In addition, the user can modify the action of system functions in ways not specifically provided for by using **ADVISE** (page 15.11). In order to encourage customizing the Interlisp environment, Interlisp includes a facility for automatically loading initialization files (or "init files") when an Interlisp system is first started. Each user can have a separate "user init file" that customizes the Interlisp environment to his/her tastes. In addition, there can be a "site init file" that applies to all users at a given physical site, setting system variables that are the same for all users such as the name of the nearest printer, etc.

The process of loading init files, also known as "greeting",, occurs when an Interlisp system created by **MAKESYS** (page 12.9) is started for the first time. The user can also explicitly invoke the greeting operation at any time via the function **GREET** (below). The process of greeting includes the following steps:

(1) Any previous greeting operation is undone. The side effects of the greeting operation are stored on a global variable as well as on the history list, thus enabling the previous greeting to be undone even if it has dropped off of the bottom of the history list.

(2) All of the items on the list **PREGREETFORMS** are evaluated.

(3) The site init file is loaded. **GREET** looks for a file by the name **{DSK}INIT.LISP**. If this is found, it is loaded. If it is not found, the system prints "**Please enter name of system init file (e.g. {server}<directory>INIT.extension):**" and waits for the user to type a file name, followed by a carriage return. If the user just types a carriage return without typing a file name, no site init file is loaded. Note: The site init file is loaded with *LDFLG* set to **SYSLOAD**, so that no file package information is saved, and nothing is printed out.

(4) The user init file is loaded. The user init file is found by using the variable **USERGREETFILES** (described below), which is normally set in the site init file. The user init file is loaded with normal file

package settings, but under errorset protection and with **PRETTYHEADER** set to **NIL** to suppress the "**FILE CREATED**" message.

(5) All of the items on the list **POSTGREETFORMS** are evaluated.

(6) A greeting is printed such as "**Hello,** *XXX*.", where *XXX* is the value of the variable **FIRSTNAME** (if non-**NIL**). The variable **GREETDATES** (below) can be set to modify this greeting for particular dates.

(GREET *NAME* **—)** [Function]

Performs the greeting for the user whose username is *NAME* (if *NAME* = **NIL**, uses the login name). When Interlisp first starts up, it performs **(GREET)**.

(GREETFILENAME *USER***)** [Function]

If *USER* is **T**, **GREETFILENAME** returns the file name of the site init file, asking the user if it doesn't exist. Otherwise, *USER* is interpreted to be a user's system name, and **GREETFILENAME** returns the file name for the user init file (if it exists).

USERGREETFILES [Variable]

USERGREETFILES specifies a series of file names to try as the user init file. The value of **USERGREETFILES** is a list, where each element is a list of litatoms. For each item in **USERGREETFILES**, the user name is substituted for the litatom **USER** and the value of **COMPILE.EXT** (page 18.13) is substituted for the litatom **COM**, and the litatoms are packed into a single file name. The first such file that is found is the user init file.

For example, suppose that the value of **USERGREETFILES** was

```
(({ERIS}< USER >LISP>INIT. COM)
 ({ERIS}< USER >LISP>INIT)
 ({ERIS}< USER >INIT. COM)
 ({ERIS}< USER >INIT))
```

If the user name was **JONES**, and the value of **COMPILE.EXT** was **DCOM**, then this would search for the files **{ERIS}<JONES>LISP>INIT.DCOM**, **{ERIS}<JONES>LISP>INIT**, **{ERIS}<JONES>INIT.DCOM**, and **{ERIS}<JONES>INIT**.

Note: The file name "specifications" in **USERGREETFILES** should be fully qualified, including all host and directory information. The directory search path (the value of **DIRECTORIES**, page 24.31) is *not* used to find the user greet files.

GREETDATES [Variable]

The value of **GREETDATES** can be used to specify special greeting messages for various dates. **GREETDATES** is a list of elements of the form (*DATESTRING* . *STRING*), e.g. **("25-DEC" . "Merry Christmas")**. The user can add entries to this list in his/her **INIT.LISP** file by using a **ADDVARS** file package command like **(ADDVARS (GREETDATES ("8-FEB" . "Happy Birthday")))**. On the specified date, the **GREET** will use the indicated salutation.

Note: Users should try to make sure that their init file is "undoable". If they use the file package command "**P**" (page 17.40) to put expressions on the file to be evaluated, they should use the "undoable" version, e.g. **/SETSYNTAX** rather than **SETSYNTAX**, etc (see page 13.26). This is so another user can come up, do a **(GREET)** and have the first user's initialization undone.

It is impossible to give a complete list of all of the variables and functions that users may want to set in their init files. The default values for system variables are chosen in the hope that they will be correct for the majority of users, so many users get along with very small init files. The following describes some of the variables that users may want to reset in their init files:

Directories

The variables **DIRECTORIES** and **LISPUSERSDIRECTORIES** (page 24.31) contain lists of directories used when searching for files. **LOGINHOST/DIR** (page 24.11) determines the default directory used when calling **CONN** with no argument.

Fonts and Printing

The variables **DISPLAYFONTDIRECTORIES**, **DISPLAYFONTEXTENSIONS**, **INTERPRESSFONTDIRECTORIES**, and **PRESSFONTWIDTHSFILES** (page 27.31) must be set before fonts can be automatically loaded from files. **DEFAULTPRINTINGHOST** (page 29.4) should be set before attempting to generate hardcopy to a printer.

Network Systems

CH.DEFAULT.ORGANIZATION and **CH.DEFAULT.DOMAIN** (page 31.8) should be set to the default NS organization and domain, when using NS network communications. If **CH.NET.HINT** (page 31.9) is set, it can reduce the amount of time spent searching for a clearinghouse.

Interlisp-D Executive

The variable **PROMPT#FLG** (page 13.22) determines whether an "event number" is printed at the beginning of every input line. The function **CHANGESLICE** (page 13.21) can be used to change the number of events that are remembered on the history list.

Copyright Notices

COPYRIGHTFLG, **COPYRIGHTOWNERS**, and **DEFAULTCOPYRIGHTOWNER** (page 17.53) control the inclusion of copyright notices on source files.

Printing Functions

****COMMENT**FLG** (page 26.43) determines how program comments are printed. **FIRSTCOL**, **PRETTYFLG**, and **CLISPIFYPRETTYFLG** (page 26.47) are among the many variables controlling how functions are pretty printed.

List Structure Editor

The variable **INITIALSLST** (page 16.76) is used when "time-stamps" are inserted in a function when it is edited. **EDITCHARACTERS** (page 16.76) is used to set the read macros used in the teletype editor.

12.2 Idle Mode

The Interlisp-D environment runs on small single-user computers, usually located in users' offices. Often, users leave their computers up and running for days, which can cause several problems. First, the phosphor in the video display screen can be perminantly marked if the same pattern is displayed for a long time (weeks). Second, if the user goes away, leaving an Interlisp-D system running, another person could possibly walk up and use the environment, taking advantage of any passwords that had been entered. To solve these problems, the Interlisp-D environment implements the concept of "idle mode."

If no keyboard or mouse action has occurred for a specified time, the Interlisp-D environment automatically enters idle mode. While idle mode is on, the display screen is blacked out, to protect the phosphor. Idle mode also runs a program to display some moving pattern on the black screen, so the screen doesn't appear broken. Usually, idle mode can be exited by pressing any key on the keyboard or mouse. However, the user can optionally specify that idle mode should erase the current password cache when it is entered, and require the next user to supply a password to exit idle mode.

Note: If either shift key is pressed while Interlisp-D is in idle mode, the current user name and the amount of time spent idling are displayed in the prompt window (which appears as long as the shift key is held down).

Idle mode can also be entered by calling the function **IDLE**, or by selecting the **Idle** menu command from the background menu (page 28.6). The **Idle** menu command has subitems that allow the user to interactively set the idle options (display program, erasing password, etc.) specified by the variable **IDLE.PROFILE**:

IDLE.PROFILE [Variable]

The value of this variable is a property list (page 3.15) which controls most aspects of idle mode. The following properties are recognized:

TIMEOUT Value is a number that determines how long (in minutes) Interlisp-D will wait before automatically entering idle mode. If **NIL**, idle mode will never be entered automatically. Default is 10 minutes.

FORGET If non-**NIL**, the user's password will be erased when idle mode is entered. Default is **NIL** (don't erase password).

Note: If the password is erased, any programs left running when idle mode is entered will fail if they try doing anything requiring passwords (such as accessing file servers).

ALLOWED.LOGINS Determines who can exit idle mode, as follows:

If the value is **NIL**, idle mode is exited without requesting login.

If the value is **LOGIN** (the default), login is required, but anyone is allowed to exit idle mode. This will overwrite the previous user's user name and password each time idle mode is exited.

If the value is one of **AUTHENTICATE**, **NS.AUTHENTICATE**, or **GV.AUTHENTICATE**, login is required and the password is checked with the net. Only allow users with accounts to exit idle mode. **NS.AUTHENTICATE** or **GV.AUTHENTICATE** specify that NS or grapevine authentication must be used, respectively. **AUTHENTICATE** indicates that either type of authentication can be tried.

If the value is a list, it should be a list of group and/or user names. The value **T** in the list means the user who was using the machine before idle mode was entered. If the value is a list, idle mode will only be exited if: (a) the new user's user name is in this list, (b) the new user is a member of a group whose name is on this list, or (c) if **T** is a member of the list, and the same user logs in with the same password.

DISPLAYFN The value of this property, which should be a function name or lambda expression, is called to display a moving pattern on the screen while in idle mode. This function is called with one argument, a window covering the whole screen. The default is **IDLE.BOUNCING.BOX** (below).

Note: Any function used as a **DISPLAYFN** should call **BLOCK** (page 23.5) frequently, so other programs can run during idle mode.

SAVEVM Value is a number that determines how long (in minutes) after idle mode is entered that **SAVEVM** (page 12.7) will be called to save the virtual memory. If **NIL**, **SAVEVM** is never called automatically from idle mode. Default is 10 minutes.

RESETVARS Value is a list of two-element lists: **(**(*VAR_1 EXP_1*) (*VAR_2 EXP_2*) ...**)**. On entering idle mode, each variable *VAR_N* is bound to the value of the corresponding expression *EXP_N*. When idle mode is exited, each variable *VAR_N* is reset to its original value.

SUSPEND.PROCESS.NAMES Value is a list of names. For each name on this list, if a process by that name is found, it will be suspended upon entering idle mode and woken upon exiting idle mode.

IDLE.FUNCTIONS [Variable]

The value of this variable determines the menu raised by selecting the **Display** subitem of the **Idle** background menu command. It should be in the format used for the **ITEMS** field of a menu (page 28.39), with the selection of an item returning the appropriate display function.

(IDLE.BOUNCING.BOX *WINDOW BOX WAIT***)** [Function]

This is the default display function used for idle mode. *BOX* is bounced about *WINDOW*, with bounces taking place every *WAIT* milliseconds. *BOX* can be a string, a bitmap, a window (whose image will be bounced about), or a list containing any number of these (which will be cycled through). *BOX* defaults to the value of the variable **IDLE.BOUNCING.BOX**, which is is initially the string **"Interlisp-D"**. *WAIT* defaults to 1000 (one second).

12.3 Saving Virtual Memory State

Interlisp storage allocation occurs within a virtual memory space that is usually much larger than the physical memory on the computer. The virtual memory is stored as a large file on the computer's hard disk, called the virtual memory file. Interlisp controls the swapping of pages between this file and the real memory, swapping in virtual memory pages as they are accessed, and swapping out pages that have been modified. At any moment, the total state of the Interlisp virtual memory is stored partially in the virtual memory file, and partially in the real physical memory.

Interlisp provides facilities for saving the total state of the virtual memory, either on the virtual memory file, or in a file on an arbitrary file device. The function **LOGOUT** is used to write all altered (dirty) pages from the real memory to the virtual memory file and stop Interlisp, so that Interlisp can be restarted from the state of the **LOGOUT**. **SAVEVM** updates the virtual memory file without stopping Interlisp, which puts the virtual memory file into a consistant state (temporarily), so it could be restarted if the system crashes. **SYSOUT** and **MAKESYS** are used to save a copy of the total virtual memory state on a file, which can be loaded into another machine to restore the Interlisp state. **VMEM.PURE.STATE** can be used to "freeze" the current state of the virtual memory, so that Interlisp will come up in that state if it is restarted.

(LOGOUT *FAST*) [Function]

Stops Interlisp, and returns control to the operating system. If Interlisp is restarted, it should come up in the same state as when the **LOGOUT** was called. **LOGOUT** will not affect the state of open files.

LOGOUT writes out all altered pages from real memory to the virtual memory file. If *FAST* is **T**, Interlisp is stopped without updating the virtual memory file. Note that after doing **LOGOUT T)** it will not be possible to restart Interlisp from the point of the **LOGOUT**, and it may not be possible to restart it at all. Typing **(LOGOUT T)** is preferable to just booting the machine, because it also does other cleanup operations (closing network connections, etc.).

If *FAST* is the litatom **?**, **LOGOUT** acts like *FLG* = **T** if the virtual memory file is consistant, otherwise it acts like *FLG* = **NIL**. This insures that the virtual memory image can be restarted as of *some* previous state, not necessarily as of the **LOGOUT**.

(SAVEVM —) [Function]

This function is similar to logging out and continuing, but faster. It takes about as long as a logout, which can be as brief as 10 seconds or so if you have already written out most of your dirty pages by virtue of being idle a while. After the **SAVEVM**, and until the pagefault handler is next forced to write out a dirty page, your virtual memory image will be continuable (as of the **SAVEVM)** should there be a system crash or other disaster.

If the system has been idle long enough (no keyboard or mouse activity), there are dirty pages to be written, and there are few enough dirty pages left to write that a **SAVEVM** would be quick, **SAVEVM** is automatically called. When **SAVEVM** is called automatically, the cursor is changed to a special cursor: SAV-ING, stored in the variable **SAVINGCURSOR**. You can control how often **SAVEVM** is automatically called by setting the following two global variables:

SAVEVMWAIT [Variable]

SAVEVMMAX [Variable]

The system will call **SAVEVM** after being idle for **SAVEVMWAIT** seconds (initially 300) if there are fewer than **SAVEVMMAX** pages dirty (initially 600). These values are fairly conservative. If you want to be extremely wary, you can set **SAVEVMWAIT** = 0 and **SAVEVMMAX** = 10000, in which case **SAVEVM** will be called the first chance available after the first dirty page has been written.

The function **SYSOUT** saves the current state of the Interlisp virtual memory on a file, known as a "sysout file", or simply a "sysout". The file package can be used to save particular function definitions and other arbitrary objects on files, but **SYSOUT** saves the *total* state of the system. This capability can be useful in many situations: for creating customized systems for other people to use, or to save a particular system state for debugging purposes. Note that a sysout file can be very large (thousands of pages), and can take a long time to create, so it is not to be done lightly. The file produced by **SYSOUT** can be loaded into the Interlisp virtual memory and restarted to restore the virtual memory to the exact state that it had when the sysout file was made. The exact method of loading a sysout depend on the implementation. For more information on loading sysout files, see the users guide for your computer.

(SYSOUT *FILE*) [Function]

Saves the current state of the Interlisp virtual memory on the file *FILE*, in a form that can be subsequently restarted. The current state of program execution is saved in the sysout file, so **(PROGN (SYSOUT 'FOO) (PRINT 'HELLO))** will cause **HELLO** to be printed after the sysout file is restarted.

SYSOUT can take a very long time (ten or fifteen minutes), particularly when storing a file on a remote file server. To display some indication that something is happening, the cursor is changed to: SYS OUT. Also, as the sysout file is being written, the cursor is inverted line by line, to show that activity is taking place, and how much of the sysout has completed. For example, after the **SYSOUT** is about two-thirds done, the cursor would look like: SYS OUT. The **SYSOUT** cursor is stored in the variable **SYSOUTCURSOR**.

If *FILE* is non-**NIL**, the variable **SYSOUTFILE** is set to the body of *FILE*. If *FILE* is **NIL**, then the value of **SYSOUTFILE** instead. Therefore, **(SYSOUT)** will save the current state on the next higher version of a file with the same name as the previous **SYSOUT**. Also, if the extension for *FILE* is not specified, the value of **SYSOUT.EXT** is used. **SYSOUT** sets **SYSOUTDATE** (page 12.13) to **(DATE)**, the time and date that the **SYSOUT** was performed.

If **SYSOUT** was not able to create the sysout file, because of disk or computer error, or because there was not enough space on the directory, **SYSOUT** returns **NIL**. Otherwise it returns the full file name of *FILE*.

Actually, **SYSOUT** "returns" twice; when the sysout file is first created, and when it is subsequently restarted. In the latter case, **SYSOUT** returns a list whose **CAR** is the full file name of *FILE*. For example, **(if (LISTP (SYSOUT 'FOO)) then (PRINT 'HELLO))** will cause **HELLO** to be printed when the sysout file is restarted, but not when **SYSOUT** is initially performed.

Note: **SYSOUT** does not save the state of any open files. **WHENCLOSE** (page 24.20) can be used to associate certain operations with open files so that when a **SYSOUT** is started up, these files will be reopened, and file pointers repositioned.

SYSOUT evaluates the expressions on **BEFORESYSOUTFORMS** before creating the sysout file. This variable initially includes expressions to: (1) Set the variables **SYSOUTDATE** and **SYSOUTFILE** as described above; (2) Default the sysout file name *FILE* according to the values of the variables **SYSOUTFILE** and **SYSOUT.EXT**, as described above; and (3) Perform any necessary operations on open files as specified by calls to **WHENCLOSE** (page 24.20).

After a sysout file is restarted (but *not* when it is initially created), **SYSOUT** evaluates the expressions on **AFTERSYSOUTFORMS**. This initially includes expressions to: (1) Perform any necessary operations on previously-opened files as specified by calls to **WHENCLOSE** (page 24.20); (2) Possibly print a message, as determined by the value of **SYSOUTGAG** (see below); and (3) Call **SETINITIALS** to reset the initials used for time-stamping (page 16.76).

SYSOUTGAG [Variable]

The value of **SYSOUTGAG** determines what is printed when a sysout file is restarted. If the value of **SYSOUTGAG** is a list, the list is evaluated, and no additional message is printed. This allows the user to print a message. If **SYSOUTGAG** is non-**NIL** and not a list, no message is printed. Finally, if **SYSOUTGAG** is **NIL** (its initial value), and the sysout file is being restarted by the same user that made the sysout originally, the user is greeted by printing the value of **HERALDSTRING** (see below) followed by a greeting message. If the sysout file was made by a different user, a message is printed, warning that the currently-loaded user init file may be incorrect for the current user (see page 12.1);

(MAKESYS *FILE NAME***)** [Function]

Used to store a new Interlisp system on the "makesys file" *FILE*. Similar to **SYSOUT**, except that before the file is made, the system is "initialized" by undoing the greet history, and clearing the display.

When the system is first started up, a "herald" is printed identifying the system, typically "**Interlisp-***XX DATE* **...**". If *NAME* is non-**NIL**, **MAKESYS** will use it instead of **Interlisp-***XX* in the herald. **MAKESYS** sets **HERALDSTRING** to the herald string printed out.

MAKESYS also sets the variable **MAKESYSDATE** (page 12.13) to **(DATE)**, i.e. the time and date the system was made.

Interlisp-D contains a routine that writes out dirty pages of the virtual memory during I/O wait, assuming that swapping has caused at least one dirty page to be written back into the virtual memory file (making it non-continuable). The frequency with which this routine runs is determined by:

BACKGROUNDPAGEFREQ [Variable]

This variable determines how often the routine that writes out dirty pages is run. The *higher* **BACKGROUNDPAGEFREQ** is set, the *greater* the time between running the dirty page writing routine. Initially it is set to 4. The lower **BACKGROUNDPAGEFREQ** is set, the less responsiveness you get at typein, so it may not be desirable to set it all the way down to 1.

(VMEM.PURE.STATE *X*) [NoSpread Function]

VMEM.PURE.STATE modifies the swapper's page replacement algorithm so that dirty pages are only written at the end of the virtual memory backing file. This "freezes" a given virtual memory state, so that Interlisp will come up in that state whenever it is restarted. This can be used to set up a "clean" environment on a pool machine, allowing each user to initialize the system simply by rebooting the computer.

The way to use **VMEM.PURE.STATE** is to set up the environment as you wish it to be "frozen," evaluate **(VMEM.PURE.STATE T)**, and then call any function that saves the virtual memory state (**LOGOUT**, **SAVEVM**, **SYSOUT**, or **MAKESYS**). From that point on, whenever the system is restarted, it will return to the state as of the saving operation. Future **LOGOUT**, **SAVEVM**, etc. operations will not reset this state.

Note: When the system is running in "pure state" mode, it uses a significant amount of the virtual memory backing file to save the "frozen" memory image, so this will reduce the amount of virtual memory space available for use.

(VMEM.PURE.STATE) returns **T** if the system is running in "pure state" mode, **NIL** otherwise.

(REALMEMORYSIZE) [Function]

Returns the number of real memory pages in the computer.

(VMEMSIZE) [Function]

Returns the number of pages in use in the virtual memory. This is the roughly the same as the number of pages required to make a sysout file on the local disk (see **SYSOUT**, page 12.8).

\LASTVMEMFILEPAGE [Variable]

Value is the total size of the virtual memory backing file. This variable is set when the system is started. It should not be set by the user.

Note: When the virtual memory expands to the point where the virtual memory backing file is almost full, a break will occur with the warning message "Your virtual memory backing file is almost full. Save your work & reload asap." When this happens, it is strongly suggested that you save any important work and reload the system. If you continue working past this point, the system will start slowing down considerably, and it will eventually stop working.

12.4 System Version Information

Interlisp-D runs on a number of different machines, with many possible hardware configurations. There have been a number of different releases of the Interlisp-D software. These facts make it difficult to answer the important question "what software/hardware environment are you running?" when reporting bugs. The following functions allow the novice to collect this information.

(PRINT-LISP-INFORMATION ***STREAM FILESTRING*****)** [NoSpread Function]

Prints out a summary of the software and hardware environment that Interlisp-D is running in, and a list of all loaded patch files:

Interlisp-D version KOTO of 10-Sep-85 08:25:46
on 1108, microcode 5658, 8191 pages,
machine 222#0.125000.34652#0 on
Interlisp-D version 9-Sep-85 18:54:29
Patch files: GCPATCH dated 11-Sep-85 10:56:37

STREAM is the stream used to print the summary. If not given, it defaults to **T**.

FILESTRING is a string used to determine what loaded files should be listed as "patch files." All file names on **LOADEDFILELST** (page 17.20) that have *FILESTRING* as a substring as listed. If *FILESTRING* is not given, it defaults to the string "PATCH".

(LISP-IMPLEMENTATION-TYPE) [Function]

Returns a string identifying the type of Interlisp implementation that is running, e.g., "Interlisp-D".

(LISP-IMPLEMENTATION-VERSION) [Function]

Returns a string identifying the version of Interlisp that is running. Currently gives the system name and date, e.g., "KOTO of 10-Sep-85 08:25:46".

This uses the variables **MAKESYSNAME** and **MAKESYSDATE** (below), so it will change if the user uses **MAKESYS** (page 12.9) to create a custom sysout file, or explicitly changes these variables.

(SOFTWARE-TYPE) [Function]

Returns a string identifying the operating system that Interlisp is running under. Currently returns the string "Interlisp-D".

(SOFTWARE-VERSION) [Function]

Returns a string identifying the version of the operating system that Interlisp is running under. Currently, this returns the date that the Interlisp-D release was originally created, so it doesn't change over **MAKESYS** or **SYSOUT**.

(MACHINE-TYPE) [Function]

Returns a string identifying the type of computer hardware that Interlisp-D is running on, i.e., "1108", "1132", "1186", etc.

(MACHINE-VERSION) [Function]

Returns a string identifying the version of the computer hardware that Interlisp-D is running on. Currently returns the microcode version and real memory size.

(MACHINE-INSTANCE) [Function]

Returns a string identifying the particular machine that Interlisp-D is running on. Currently returns the machine's NS address.

(SHORT-SITE-NAME) [Function]

Returns a short string identifying the site where the machine is located. Currently returns **(ETHERHOSTNAME)** (if non-**NIL**) or the string "unknown".

(LONG-SITE-NAME) [Function]

Returns a long string identifying the site where the machine is located. Currently returns the same as **SHORT-SITE-NAME**.

SYSOUTDATE [Variable]

Value is set by **SYSOUT** (page 12.8) to the date before generating a virtual memory image file.

MAKESYSDATE [Variable]

Value is set by **MAKESYS** (page 12.9) to the date before generating a virtual memory image file.

MAKESYSNAME [Variable]

Value is a litatom identifying the release name of the current Interlisp-D system, e.g., **KOTO**.

(SYSTEMTYPE) [Function]

The **SYSTEMTYPE** function is intended to allow programmers to write system-dependent code. **SYSTEMTYPE** returns a litatom corresponding to the implementation of Interlisp: **D** (for Interlisp-D), **TOPS-20**, **TENEX**, **JERICO**, or **VAX**.

In Interlisp-D (and Interlisp-10), **(SELECTQ (SYSTEMTYPE) ...)** expressions are expanded at compile time so that this is an effective way to perform conditional compilation.

(MACHINETYPE) [Function]

Returns the type of machine that Interlisp-D is running on: either **DORADO** (for the Xerox 1132), **DOLPHIN** (for the Xerox 1100), or **DANDELION** (for the Xerox 1108).

12.5 Date And Time Functions

(DATE ***FORMAT*****)** [Function]

Returns the current date and time as a string with format "*DD-MM-YY HH:MMM:SS*", where *DD* is day, *MM* is month, *YY* year, *HH* hours, *MMM* minutes, *SS* seconds, e.g., "** 7-Jun-85 15:49:34**".

If *FORMAT* is a date format as returned by **DATEFORMAT** (below), it is used to modify the format of the date string returned by **DATE**.

(IDATE ***STR*****)** [Function]

STR is a date and time string. **IDATE** returns *STR* converted to a number such that if $DATE_1$ is before (earlier than) $DATE_2$, then **(IDATE** $DATE_1$**)** < **(IDATE** $DATE_2$**)**. If *STR* is **NIL**, the current date and time is used.

Note that different Interlisp implementations can have different internal date formats. However, **IDATE** always has the essential property that **(IDATE** *X***)** is less than **(IDATE** *Y***)** if *X* is before *Y*, and **(IDATE (GDATE** *N***))** equals *N*. Programs which do arithmetic other than numerical comparisons between **IDATE** numbers may not work when moved from one implementation to another.

Generally, it is possible to increment an **IDATE** number by an integral number of days by computing a "1 day" constant, the difference between two convenient **IDATE's**, e.g. **(IDIFFERENCE (IDATE " 2-JAN-80 12:00") (IDATE " 1-JAN-80 12:00"))**. This "1 day" constant can be evaluated at compile time.

IDATE is guaranteed to accept as input the dates that **DATE** will output. It will ignore the parenthesized day of the week (if any). **IDATE** also correctly handles time zone specifications for those time zones registered in the list **TIME.ZONES** (page 12.15).

(GDATE *DATE FORMAT —***)** [Function]

Like **DATE**, except that *DATE* can be a number in internal date and time format as returned by **IDATE**. If *DATE* is **NIL**, the current time and date is used.

(DATEFORMAT KEY_1 ... KEY_N**)** [NLambda NoSpread Function]

DATEFORMAT returns a date format suitable as a parameter to **DATE** and **GDATE**. KEY_1 ... KEY_N are a set of keywords (unevaluated). Each keyword affects the format of the date independently (except for **SLASHES** and **SPACES**). If the date returned by **(DATE)** with the default formatting was **" 7-Jun-85 15:49:34"**, the keywords would affect the formatting as follows:

NO.DATE Doesn't include the date information, e.g. **"15:49:34"**.

NUMBER.OF.MONTH Shows the month as a number instead of a name, e.g. **" 7-06-85 15:49:34"**.

YEAR.LONG Prints the year using four digits, e.g. **" 7-Jun-1985 15:49:34"**.

SLASHES Separates the day, month, and year fields with slashes, e.g. **" 7/Jun/85 15:49:34"**.

SPACES Separates the day, month, and year fields with spaces, e.g. **" 7 Jun 85 15:49:34"**.

NO.LEADING.SPACES By default, the day field will always be two characters long. If **NO.LEADING.SPACES** is specified, the day field will be one character for dates earlier than the 10th, e.g. **"7-Jun-85 15:49:34"** instead of **" 7-Jun-85 15:49:34"**.

NO.TIME Doesn't include the time information, e.g. **" 7-Jun-85"**.

TIME.ZONE Includes the time zone in the time specification, e.g. **" 7-Jun-85 15:49:34 PDT"**.

NO.SECONDS Doesn't include the seconds, e.g. **" 7-Jun-85 15:49"**.

DAY.OF.WEEK Includes the day of the week in the time specification, e.g. " **7-Jun-85 15:49:34 PDT (Friday)"**.

DAY.SHORT If **DAY.OF.WEEK** is specified to include the day of the week, the week day is shortened to the first three letters, e.g. " **7-Jun-85 15:49:34 PDT (Fri)"**. Note that **DAY.SHORT** has no effect unless **DAY.OF.WEEK** is also specified.

(CLOCK *N* **—)** [Function]

If *N* = 0, **CLOCK** returns the current value of the time of day clock i.e., the number of milliseconds since last system start up.

If *N* = 1, returns the value of the time of day clock when the user started up this Interlisp, i.e., difference between **(CLOCK 0)** and **(CLOCK 1)** is number of milliseconds (real time) since this Interlisp system was started.

If *N* = 2, returns the number of milliseconds of *compute* time since user started up this Interlisp (garbage collection time is subtracted off).

If *N* = 3, returns the number of milliseconds of compute time spent in garbage collections (all types).

(SETTIME *DT***)** [Function]

Sets the internal time-of-day clock. If *DT* = **NIL**, **SETTIME** attempts to get the time from the communications net; if it fails, the user is prompted for the time. If *DT* is a string in a form that **IDATE** recognizes, it is used to set the time.

The following variables are used to interpret times in different time zones. **\TimeZoneComp**, **\BeginDST**, and **\EndDST** are normally set automatically if your machine is connected to a network with a time server. For standalone machines, it may be necessary to set them by hand (or in your init file, see page 12.1) if you are not in the Pacific time zone.

TIME.ZONES [Variable]

Value is an association list that associates time zone specifications (PDT, EST, GMT, etc.) with the number of hours west of Greenwich (negative if east). If the time zone specification is a single letter, it is appended to "DT" or "ST" depending on whether daylight saving time is in effect. Initially set to:

((8 . P) (7 . M) (6 . C) (5 . E) (0 . GMT))

This list is used by **DATE** and **GDATE** when generating a date with the **TIME.ZONE** format is specified, and by **IDATE** when parsing dates.

\TimeZoneComp [Variable]

This variable should be initialized to the number of hours west of Greenwich (negative if east). For the U.S. west coast it is 8. For the east coast it is 5.

\BeginDST [Variable]

\EndDST [Variable]

\BeginDST is the day of the year on or before which Daylight Savings Time takes effect (i.e., the Sunday on or immediately preceding this day); **\EndDST** is the day on or before which Daylight Savings Time ends. Days are numbered with 1 being January 1, and counting the days as for a leap year. In the USA where Daylight Savings Time is observed, **\BeginDST** = 121 and **\EndDST** = 305. In a region where Daylight Savings Time is not observed at all, set **\BeginDST** to 367.

12.6 Timers and Duration Functions

Often one needs to loop over some code, stopping when a certain interval of time has passed. Some systems provide an "alarm clock" facility, which provides an asynchronous interrupt when a time interval runs out. This is not particularly feasible in the current Interlisp-D environment, so the following facilities are supplied for efficiently testing for the expiration of a time interval in a loop context.

Three functions are provided: **SETUPTIMER**, **SETUPTIMER.DATE**, and **TIMEREXPIRED?**. Also several new i.s.oprs have been defined: **forDuration**, **during**, **untilDate**, **timerUnits**, **usingTimer**, and **resourceName** (reasonable variations on upper/lower case are permissible).

These functions use an object called a timer, which encodes a future clock time at which a signal is desired. A timer is constructed by the functions **SETUPTIMER** and **SETUPTIMER.DATE**, and is created with a basic clock "unit" selected from among **SECONDS**, **MILLISECONDS**, or **TICKS**. The first two timer units provide a machine/system independent interface, and the latter provides access to the "real", basic strobe unit of the machine's clock on which the program is running. The default unit is **MILLISECONDS**.

Currently, the **TICKS** unit is a function of the particular machine that Interlisp-D is running on. The Xerox 1132 has about 1680 ticks per millisecond; the Xerox 1108 has about 34.746 ticks per millisecond; the Xerox 1185 and 1186 have about 62.5 ticks per millisecond. The advantage of using **TICKS** rather than one of the uniform interfaces is primarily speed; e.g., it may take over 400 microseconds to read the milliseconds clock (a software facility that uses the real clock), whereas reading the real clock itself may take less than ten microseconds. The disadvantage of the **TICKS** unit is its short roll-over interval (about 20 minutes) compared to the **MILLISECONDS** roll-over interval (about two weeks), and also the dependency on particular machine parameters.

(SETUPTIMER *INTERVAL OldTimer? timerUnits intervalUnits***)** [Function]

SETUPTIMER returns a timer that will "go off" (as tested by **TIMEREXPIRED?**) after a specified time-interval measured from the current clock time. **SETUPTIMER** has one required and three optional arguments:

INTERVAL must be a integer specifying how long an interval is desired. *timerUnits* specifies the units of measure for the interval (defaults to **MILLISECONDS**).

If *OldTimer?* is a timer, it will be reused and returned, rather than allocating a new timer. *intervalUnits* specifies the units in which the *OldTimer?* is expressed (defaults to the value of *timerUnits*.

(SETUPTIMER.DATE *DTS OldTimer?***)** [Function]

SETUPTIMER.DATE returns a timer (using the **SECONDS** time unit) that will "go off" at a specified date and time. *DTS* is a Date/Time string such as **IDATE** accepts (page 12.14). If *OldTimer?* is a timer, it will be reused and returned, rather than allocating a new timer.

SETUPTIMER.DATE operates by first subtracting **(IDATE)** from **(IDATE** *DTS***)**, so there may be some large integer creation involved, even if *OLDTIMER?* is given.

(TIMEREXPIRED? *TIMER ClockValue.or.timerUnits***)** [Function]

If *TIMER* is a timer, and *ClockValue.or.timerUnits* is the time-unit of *TIMER*, **TIMEREXPIRED?** returns true if *TIMER* has "gone off".

ClockValue.or.timerUnits can also be a timer, in which case **TIMEREXPIRED?** compares the two timers (which must be in the same timer units). If **X** and **Y** are timers, then **(TIMEREXPIRED? X Y)** is true if **X** is set for an *earlier* time than **Y**.

There are a number of i.s.oprs that make it easier to use timers in iterative statements (page 9.9). These i.s.oprs are given below in the "canonical" form, with the second "word" capitalized, but the all-caps and all-lower-case versions are also acceptable.

forDuration *INTERVAL* [I.S. Operator]

during *INTERVAL* [I.S. Operator]

INTERVAL is an integer specifying an interval of time during which the iterative statement will loop.

timerUnits *UNITS* [I.S. Operator]

UNITS specifies the time units of the *INTERVAL* specified in **forDuration**.

untilDate *DTS* [I.S. Operator]

DTS is a Date/Time string (such as **IDATE** accepts) specifying when the iterative statement should stop looping.

usingTimer *TIMER* [I.S. Operator]

If **usingTimer** is given, *TIMER* is reused as the timer for **forDuration** or **untilDate**, rather than creating a new timer. This can reduce allocation if one of these i.s.oprs is used within another loop.

resourceName *RESOURCE* [I.S. Operator]

RESOURCE specifies a resource name to be used as the timer storage (see page 17.24). If *RESOURCE* = **T**, it will be converted to an internal name.

Some examples:

```
(during 6MONTHS timerUnits 'SECONDS
 until (TENANT-VACATED? HouseHolder)
 do (DISMISS <for-about-a-day>)
   (HARRASS HouseHolder)
 finally (if (NOT (TENANT-VACATED? HouseHolder))
      then (EVICT-TENANT HouseHolder)))
```

This example shows that how is is possible to have two termination condition: (1) when the time interval of **6MONTHS** has elapsed, or (2) when the predicate **(TENANT-VACATED? HouseHolder)** becomes true. Note that the "finally" clause is executed regardless of which termination condition caused it.

Also note that since the millisecond clock will "roll over" about every two weeks, "**6MONTHS**" wouldn't be an appropriate interval if the timer units were the default case, namely **MILLISECONDS**.

```
(do (forDuration (CONSTANT (ITIMES 10 24 60 60 1000))
   do (CARRY.ON.AS.USUAL)
   finally (PROMPTPRINT "Have you had your 10-day
check-up?")))
```

This infinite loop breaks out with a warning message every 10 days. One could question whether the millisecond clock, which is used by default, is appropriate for this loop, since it rolls-over about every two weeks.

```
(SETQ \RandomTimer (SETUPTIMER 0))
(untilDate "31-DEC-83 23:59:59" usingTimer \RandomTimer
 when (WINNING?) do (RETURN)
 finally (ERROR "You've been losing this whole year!"))
```

Here we see a usage of an explicit date for the time interval; also, the user has squirreled away some storage (as the value of **\RandomTimer**) for use by the call to **SETUPTIMER** in this loop.

```
(forDuration SOMEINTERVAL
 resourceName \INNERLOOPBOX
 timerunits 'TICKS
 do (CRITICAL.INNER.LOOP))
```

For this loop, the user doesn't want any **CONS**ing to take place, so **\INNERLOOPBOX** will be defined as a resource which "caches" a timer cell (if it isn't already so defined), and wraps the entire statement in a **WITH-RESOURCES** call. Furthermore, he has specified a time unit of **TICKS**, for lower overhead in this critical inner loop. In fact specifying a **resourceName** of **T** would have been the same as specifying it to be **\ForDurationOfBox**; this is just a simpler way to specify that a resource is wanted, without having to think up a name.

12.7 Resources

Interlisp is based on the use of a storage-management system which allocates memory space for new data objects, and automatically reclaims the space when no longer in use. More generally, Interlisp manages shared "resources", such as files, semaphors for processes, etc. The protocols for allocating and freeing such resources resemble those of ordinary storage management.

Sometimes users need to explicitly manage the allocation of resources. They may desire the efficiency of explicit reclamation of certain temporary data; or it may be expensive to initialize a complex data object; or there may be an application that must not allocate new cells during some critical section of code.

The file package type **RESOURCES** is available to help with the definition and usage of such classes of data; the definition of a **RESOURCE** specifies prototype code to do the basic management operations. The filepkg command **RESOURCES** (page 17.39) is used to save such definitions on files, and **INITRESOURCES** (page 17.39) causes the initialization code to be output.

The basic needs of resource management are (1) obtaining a data item from the Lisp memory management system and configuring it to be a totally new instance of the resource in question, (2) freeing up an instance which is no longer needed, (3) getting an instance of the resource for temporary usage [whether "fresh" or a formerly freed-up instance], and (4) setting up any prerequisite global data structures and variables. A resources definition consists of four "methods": **INIT**, **NEW**, **GET**, and **FREE**; each "method" is a form that will specialize the definition for four corresponding user-level macros **INITRESOURCE**, **NEWRESOURCE**, **GETRESOURCE**, and **FREERESOURCE**. **PUTDEF** is used to make a resources definition, and the four components are specified in a proplist:

```
(PUTDEF
  'RESOURCENAME
  'RESOURCES
  '(NEW NEW-INSTANCE-GENERATION-CODE
    FREE FREEING-UP-CODE
    GET GET-INSTANCE-CODE
    INIT INITIALIZATION-CODE))
```

Each of the *xxx-CODE* forms is a form that will appear as if it were the body of a substitution macro definition for the corresponding macro [see the discussion on the macros below].

12.7.1 A Simple Example

Suppose one has several pieces of code which use a 256-character string as a scratch string. One could simply generate a new string each time, but that would be inefficient if done repeatedly. If the user can guarantee that there are no re-entrant uses of the scratch string, then it could simply be stored in a global variable. However, if the code might be re-entrant on occasion, the program has to take precautions that two programs do not use the same scratch string at the same time. [Note: this consideration becomes very important in a multi-process environment. It is hard to guarantee that two processes won't be running the same code at the same time, without using elaborate locks.] A typical tactic would be to store the scratch string in a global variable, and set the variable to **NIL** whenever the string is in use (so that re-entrant usages would know to get a "new" instance). For example, assuming the global variable **TEMPSTRINGBUFFER** is initialized to **NIL**:

```
[DEFINEQ (WITHSTRING NIL
 (PROG ((BUF (OR (PROG1 TEMPSTRINGBUFFER
                 (SETQ TEMPSTRINGBUFFER NIL))
```

```
        (ALLOCSTRING 256))))
  ... use the scratch string in the variable BUF ...
  (SETQ TEMPSTRINGBUFFER BUF)
  (RETURN]
```

Here, the basic elements of a "resource" usage may be seen: (1) a call **(ALLOCSTRING 256)** allocates fresh instances of "buffer", (2) after usage is completed the instance is returned to the "free" state, by putting it back in the global variable **TEMPSTRINGBUFFER** where a subsequent call will find it, (3) the prog-binding of **BUF** will get an existing instance of a string buffer if there is one -- otherwise it will get a new instance which will later be available for reuse, and (4) some initialization is performed before usage of the resource (in this case, it is the setting of the global variable **TEMPSTRINGBUFFER**).

Given the following resources definition:

```
(PUTDEF
  'STRINGBUFFER
  'RESOURCES
  '(NEW  (ALLOCSTRING 256)
    FREE (SETQ TEMPSTRINGBUFFER (PROG1 . ARGS))
    GET  (OR (PROG1 TEMPSTRINGBUFFER
               (SETQ TEMPSTRINGBUFFER NIL))
             (NEWRESOURCE TEMPSTRINGBUFFER)))
    INIT (SETQ TEMPSTRINGBUFFER NIL)))
```

we could then redo the example above as

```
(DEFINEQ (WITHSTRING NIL
  (PROG ((BUF (GETRESOURCE STRINGBUFFER)))
    ... use the string in the variable BUF ...
    (FREERESOURCE STRINGBUFFER BUF)
    (RETURN]
```

The advantage of doing the coding this way is that the resource management part of **WITHSTRING** is fully contained in the expansions of **GETRESOURCE** and **FREERESOURCE**, and thus there is no confusion between what is **WITHSTRING** code and what is resource management code. This particuar advantage will be multiplied if there are other functions which need a "temporary" string buffer; and of course, the resultant modularity makes it much easier to contemplate minor variations on, as well as multiple clients of, the **STRINGBUFFER** resource.

In fact, the scenario just shown above in the **WITHSTRING** example is so commonly useful that an abbreviation has been added; if a resources definition is made with *only* a **NEW** method, then appropriate **FREE**, **GET**, and **INIT** methods will be inferred, along with a coordinated globalvar, to be parallel to the above definition. So the above definition could be more simply written

```
(PUTDEF 'STRINGBUFFER
    'RESOURCES
    '(NEW (ALLOCSTRING 256)))
```

and every thing would work the same.

The macro **WITH-RESOURCES** simplifies the common scoping case, where at the beginning of some piece of code, there are one or more **GETRESOURCE** calls the results of which are each bound to a lambda variable; and at the ending of that code a **FREERESOURCE** call is done on each instance. Since the resources are locally bound to variables with the same name as the resource itself, the definition for **WITHSTRING** then simplifies to

```
(DEFINEQ (WITHSTRING NIL
  (WITH-RESOURCES (STRINGBUFFER)
   ... use the string in the variable STRINGBUFFER ...]
```

12.7.2 Trade-offs in More Complicated Cases

This simple example presumes that the various functions which use the resource are generally not re-entrant. While an occasional re-entrant use will be handled correctly (another example of the resource will simply be created), if this were to happen too often, then many of the resource requests will create and throw away new objects, which defeats one of the major purposes of using resources. A slightly more complex **GET** and **FREE** method can be of much benefit in maintaining a pool of available resources; if the resource were defined to maintain a list of "free" instances, then the **GET** method could simply take one off the list and the **FREE** method could just push it back onto the list. In this simple example, the **SETQ** in the **FREE** method defined above would just become a "push", and the first clause of the **GET** method would just be **(pop TEMPSTRINGBUFFER)**

A word of caution: if the datatype of the resource is something very small that Interlisp system is "good" at allocating and reclaiming, then explicit user storage management will probably not do much better than the combination of cons/createcell and the garbage collector. This would especially be so if more complicated **GET** and **FREE** methods were to be used, since their overhead would be closer to that of the built-in system facilities. Finally, it must be considered whether retaining multiple instances of the resource is a net gain; if the re-entrant case is truly rare, it may be more worthwhile to retain at most one instance, and simply let the instances created by the rarely-used case be reclaimed in the normal course of garbage collection.

12.7.3 Macros for Accessing Resources

Four user-level macros are defined for accessing resources:

(NEWRESOURCE *RESOURCENAME . ARGS***)** [Macro]

(FREERESOURCE *RESOURCENAME . ARGS***)** [Macro]

(GETRESOURCE *RESOURCENAME . ARGS***)** [Macro]

(INITRESOURCE *RESOURCENAME . ARGS***)** [Macro]

Each of these macros behave as if they were defined as a substitution macro of the form

((*RESOURCENAME . ARGS***)** *MACROBODY***)**

where the expression *MACROBODY* is selected by using the "code" supplied by the corresponding method from the *RESOURCENAME* definition.

Note that it is possible to pass "arguments" to the user's resource allocation macros. For example, if the **GET** method for the resource **FOO** is **(GETFOO . ARGS)**, then **(GETRESOURCE FOO X Y)** is transformed into **(GETFOO X Y)**. This form was used in the **FREE** method of the **STRINGBUFFER** resource described above, to pass the old **STRINGBUFFER** object to be freed.

(WITH-RESOURCES (*$RESOURCE_1$ $RESOURCE_2$ ...***)** *$FORM_1$ $FORM_2$ ...***)** [Macro]

The **WITH-RESOURCES** macro binds lambda variables of the same name as the resources (for each of the resources *$RESOURCE_1$*, *$RESOURCE_2$*, etc.) to the result of the **GETRESOURCE** macro; then executes the forms *$FORM_1$*, *$FORM_2$*, etc., does a **FREERESOURCE** on each instance, and returns the value of the last form (evaluated and saved before the **FREERESOURCE**s).

Note: **(WITH-RESOURCES** *RESOURCE* **...)** is interpreted the same as **(WITH-RESOURCES (***RESOURCE***) ...)**. Also, the singular name **WITH-RESOURCE** is accepted as a synonym for **WITH-RESOURCES**.

12.7.4 Saving Resources in a File

Resources definitions may be saved on files using the **RESOURCES** file package command (page 17.39). Typically, one only needs the full definition available when compiling or interpreting the code, so it is appropriate to put the file package command in a **(DECLARE: EVAL@COMPILE DONTCOPY ...)** declaration, just as one might do for a **RECORDS** declaration. But just as certain record declarations need *some* initialization in the run-time environment, so do most resources. This initialization is specified by the resource's **INIT** method, which is executed automatically when the resource is defined by the **PUTDEF** output by the **RESOURCES** command. However, if the **RESOURCES** command is in a **DONTCOPY** expression and thus is not included in the compiled file, then it is necessary to include a separate **INITRESOURCES** command (page 17.39) in the filecoms to insure that the resource is properly initialized.

12.8 Pattern Matching

Interlisp provides a fairly general pattern match facility that allows the user to specify certain tests that would otherwise be clumsy to write, by giving a pattern which the datum is supposed to match. Essentially, the user writes "Does the (expression) X look like (the pattern) P?" For example, **(match X with (& 'A -- 'B))** asks whether the second element of **X** is an **A**, and the last element a **B**. The implementation of the matching is performed by computing (once) the equivalent Interlisp expression which will perform the indicated operation, and substituting this for the pattern, and *not* by invoking each time a general purpose capability such as that found in FLIP or PLANNER. For example, the translation of **(match X with (& 'A -- 'B))** is:

```
(AND (EQ (CADR X) 'A)
  (EQ (CAR (LAST (CDDR X))) 'B))
```

Thus the pattern match facility is really a pattern match compiler, and the emphasis in its design and implementation has been more on the efficiency of object code than on generality and sophistication of its matching capabilities. The goal was to provide a facility that could and would be used even where efficiency was paramount, e.g., in inner loops. As a result, the pattern match facility does not contain (yet) some of the more esoteric features of other pattern match languages, such as repeated patterns, disjunctive and conjunctive patterns, recursion, etc. However, the user can be confident that what facilities it does provide will result in Interlisp expressions comparable to those he would generate by hand. Wherever possible, already existing Interlisp functions are used in the translation, e.g., the translation of **($ 'A $)** uses **MEMB**, **($ ('A $) $)** uses **ASSOC**, etc.

The syntax for pattern match expressions is **(match** *FORM* **with** *PATTERN***)**, where *PATTERN* is a list as described below. If *FORM* appears more than once in the translation, and it is not either a variable, or an expression that is easy to (re)compute, such as **(CAR Y)**, **(CDDR Z)**, etc., a dummy variable will be generated and bound to the value of *FORM* so that *FORM* is not evaluated a multiple number of times. For example, the translation of **(match (FOO X) with ($ 'A $))** is simply **(MEMB 'A (FOO X))**, while the translation of **(match (FOO X) with ('A 'B --))** is:

```
[PROG ($$2)
 (RETURN
  (AND (EQ (CAR (SETQ $$2 (FOO X)))
         'A)
       (EQ (CADR $$2) 'B]
```

In the interests of efficiency, the pattern match compiler assumes that all lists end in **NIL**, i.e., there are no **LISTP** checks inserted in the translation to check tails. For example, the translation of **(match X with ('A & --))** is **(AND (EQ (CAR X) (QUOTE A)) (CDR X))**, which will match with **(A B)** as well as **(A . B)**. Similarly, the pattern match compiler does not insert **LISTP** checks on elements, e.g., **(match X with (('A --) --))** translates simply as **(EQ (CAAR X) 'A)**, and **(match X with (($1 $1 --) --))** as **(CDAR X)**. Note that the user can explicitly insert **LISTP** checks himself by using **@**, as described below, e.g., **(match X with (($1 $1 --)@LISTP --))** translates as **(CDR (LISTP (CAR X)))**.

Note: The insertion of **LISTP** checks for *elements* is controlled by the variable **PATLISTPCHECK**. When **PATLISTPCHECK** is **T**, **LISTP** checks are inserted, e.g., **(match X with (('A --) --))** translates as: **(EQ (CAR (LISTP (CAR (LISTP X)))) 'A)**. **PATLISTPCHECK** is initially **NIL**. Its value can be changed within a particular function by using a local CLISP declaration (see page 21.13).

Note: Pattern match expressions are translated using the DWIM and CLISP facilities, using all CLISP declarations in effect (standard/fast/undoable) (see page 21.12).

12.8.1 Pattern Elements

A pattern consists of a list of pattern elements. Each pattern element is said to match either an element of a data structure or a segment. For example, in the editor's pattern matcher, "--" (page 16.19) matches any arbitrary segment of a list, while **&** or a subpattern match only one element of a list. Those patterns which may match a segment of a list are called *segment* patterns; those that match a single element are called *element* patterns.

12.8.2 Element Patterns

There are several types of element patterns, best given by their syntax:

$1 or & — Matches an arbitrary element of a list.

'*EXPRESSION* — Matches only an element which is equal to the given expression e.g., **'A**, **'(A B)**.

EQ, **MEMB**, and **ASSOC** are automatically used in the translation when the quoted expression is atomic, otherwise **EQUAL**, **MEMBER**, and **SASSOC**.

= ***FORM*** — Matches only an element which is **EQUAL** to the value of *FORM*, e.g., **=X**, **=(REVERSE Y)**.

== ***FORM*** — Same as **=**, but uses an **EQ** check instead of **EQUAL**.

ATOM — The treatment depends on setting of **PATVARDEFAULT**. If **PATVARDEFAULT** is **'** or **QUOTE**, same as **'***ATOM*. If **PATVARDEFAULT** is **=** or **EQUAL**, same as **=***ATOM*. If **PATVARDEFAULT** is **==** or **EQ**, same as **==***ATOM*. If **PATVARDEFAULT** is **←** or **SETQ**, same as *ATOM*←&. **PATVARDEFAULT** is initially **'**.

PATVARDEFAULT can be changed within a particular function by using a local CLISP declaration (see page 21.13).

Note: numbers and strings are always interpreted as though **PATVARDEFAULT** were **=**, regardless of its setting. **EQ**, **MEMB**, and **ASSOC** are used for comparisons involving small integers.

(*PATTERN*$_1$ ... *PATTERN*$_N$) — Matches a list which matches the given patterns, e.g., **(& &)**, **(-- 'A)**.

ELEMENT-PATTERN*@*FN — Matches an element if *ELEMENT-PATTERN* matches it, and *FN* (name of a function or a **LAMBDA** expression) applied to that element returns non-**NIL**. For example, **&@NUMBERP** matches a number and **('A --)@FOO** matches a list whose first element is **A**, and for which **FOO** applied to that list is non-**NIL**.

For "simple" tests, the function-object is applied before a match is attempted with the pattern, e.g., **((-- 'A --)@LISTP --)** translates as **(AND (LISTP (CAR X)) (MEMB 'A (CAR X)))**, not the other way around. *FN* may also be a *FORM* in terms of the variable **@**, e.g., **&@(EQ @ 3)** is equivalent to **=3**.

***** — Matches any arbitrary element. If the entire match succeeds, the element which matched the * will be returned as the value of the match.

Note: Normally, the pattern match compiler constructs an expression whose value is guaranteed to be non-**NIL** if the match succeeds and **NIL** if it fails. However, if a * appears in the pattern, the expression generated could also return **NIL** if the match succeeds and * was matched to **NIL**. For example, **(match X with ('A * --))** translates as **(AND (EQ (CAR X) 'A) (CADR X))**, so if **X** is equal to **(A NIL B)** then **(match X with ('A * --))** returns **NIL** even though the match succeeded.

~ELEMENT-PATTERN Matches an element if the element is *not* matched by *ELEMENT-PATTERN*, e.g., **~'A**, **~=X**, **~(-- 'A --)**.

(*ANY* *ELEMENT-PATTERN ELEMENT-PATTERN* **...)** Matches if any of the contained patterns match.

12.8.3 Segment Patterns

$ or **--** Matches any segment of a list (including one of zero length).

The difference between **$** and **--** is in the type of search they generate. For example, **(match X with ($ 'A 'B $))** translates as **(EQ (CADR (MEMB 'A X)) 'B)**, whereas **(match X with (-- 'A 'B $))** translates as:

```
[SOME X (FUNCTION (LAMBDA ($$2 $$1)
 (AND (EQ $$2 'A)
   (EQ (CADR $$1) 'B]
```

Thus, a paraphrase of **($ 'A 'B $)** would be "Is the element following the *first* **A** a **B**?", whereas a paraphrase of **(-- 'A 'B $)** would be "Is there *any* **A** immediately followed by a **B**?" Note that the pattern employing **$** will result in a more efficient search than that employing **--**. However, **($ 'A 'B $)** will not match with **(X Y Z A M O A B C)**, but **(-- 'A 'B $)** will.

Essentially, once a pattern following a **$** matches, the **$** never resumes searching, whereas **--** produces a translation that will always continue searching until there is no possibility of success. However, if the pattern match compiler can deduce from the pattern that continuing a search after a particular failure cannot possibly succeed, then the translations for both **--** and **$** will be the same. For example, both **(match X with ($ 'A $3 $))** and **(match X with (-- 'A $3 --))** translate as **(CDDDR (MEMB (QUOTE A) X))**, because if there are not three elements following the first **A**, there certainly will not be three elements following subsequent **A**'s, so there is no reason to continue searching, even for **--**. Similarly, **($ 'A $ 'B $)** and **(-- 'A -- 'B --)** are equivalent.

$2, **$3**, etc. Matches a segment of the given length. Note that **$1** is not a segment pattern.

!ELEMENT-PATTERN Matches any segment which *ELEMENT-PATTERN* would match as a list. For example, if the value of **FOO** is **(A B C)**, **! = FOO** will match the segment ... **A B C** ... etc.

Note: Since **!** appearing in front of the last pattern specifies a match with some *tail* of the given expression, it also makes sense in this case for a **!** to appear in front of a pattern that can only match with an atom, e.g., **($2 !'A)** means match if **CDDR** of the expression is the atom **A**. Similarly, **(match X with ($! 'A))** translates to **(EQ (CDR (LAST X)) 'A)**.

!ATOM treatment depends on setting of **PATVARDEFAULT**. If **PATVARDEFAULT** is **'** or **QUOTE**, same as **!'***ATOM* (see above

discussion). If **PATVARDEFAULT** is **=** or **EQUAL**, same as **! =***ATOM*. If **PATVARDEFAULT** is **= =** or **EQ**, same as **! = =***ATOM*. If **PATVARDEFAULT** is **←** or **SETQ**, same as *ATOM***←$**.

. The atom "." is treated *exactly* like "!". In addition, if a pattern ends in an atom, the "." is first changed to "!", e.g., **($1 . A)** and **($1 ! A)** are equivalent, even though the atom "." does not explicitly appear in the pattern.

One exception where "." is not treated like "!": "." preceding an assignment does not have the special interpretation that "!" has preceding an assignment (see below). For example, **(match X with ('A . FOO←'B))** translates as:

```
(AND (EQ (CAR X) 'A)
   (EQ (CDR X) 'B)
   (SETQ FOO (CDR X)))
```

but **(match X with ('A ! FOO←'B))** translates as:

```
(AND (EQ (CAR X) 'A)
   (NULL (CDDR X))
   (EQ (CADR X) 'B)
   (SETQ FOO (CDR X)))
```

SEGMENT-PATTERN@FUNCTION-OBJECT Matches a segment if the segment-pattern matches it, and the function object applied to the corresponding segment (as a list) returns non-**NIL**. For example, **($@CDDR 'D $)** matches **(A B C D E)** but not **(A B D E)**, since **CDDR** of **(A B)** is **NIL**.

Note: an @ pattern applied to a segment will require *computing* the corresponding structure (with **LDIFF**) each time the predicate is applied (except when the segment in question is a tail of the list being matched).

12.8.4 Assignments

Any pattern element may be preceded by "*VARIABLE*←", meaning that if the match succeeds (i.e., everything matches), *VARIABLE* is to be set to the thing that matches that pattern element. For example, if **X** is **(A B C D E)**, **(match X with ($2 Y←$3))** will set **Y** to **(C D E)**. Note that assignments are not performed until the entire match has succeeded, so assignments cannot be used to specify a search for an element found earlier in the match, e.g., **(match X with (Y←$1 =Y --))** will *not* match with **(A A B C ...)**, unless, of course, the value of **Y** was **A** before the match started. This type of match is achieved by using place-markers, described below.

If the variable is preceded by a **!**, the assignment is to the *tail* of the list as of that point in the pattern, i.e., that portion of the list matched by the remainder of the pattern. For example, if **X** is **(A B C D E)**, **(match X with ($!Y←'C 'D $))** sets **Y** to **(C D E)**, i.e., **CDDR** of **X**. In other words, when ! precedes an assignment, it acts as a modifier to the ←, and has no effect whatsoever on the pattern itself, e.g., **(match X with ('A 'B))** and **(match X with ('A !FOO←'B))** match identically, and in the latter case, **FOO** will be set to **CDR** of **X**.

Note: **←PATTERN-ELEMENT* and *!*←PATTERN-ELEMENT* are acceptable, e.g., **(match X with ($ 'A *←('B --) --))** translates as:

```
[PROG ($$2) (RETURN
 (AND (EQ (CAADR (SETQ $$2 (MEMB 'A X))) 'B)
   (CADR $$2]
```

12.8.5 Place-Markers

Variables of the form *#N*, *N* a number, are called place-markers, and are interpreted specially by the pattern match compiler. Place-markers are used in a pattern to mark or refer to a particular pattern element. Functionally, they are used like ordinary variables, i.e., they can be assigned values, or used freely in forms appearing in the pattern, e.g., **(match X with (#1←$1 =(ADD1 #1)))** will match the list **(2 3)**. However, they are not really variables in the sense that they are not bound, nor can a function called from within the pattern expect to be able to obtain their values. For convenience, regardless of the setting of **PATVARDEFAULT**, the first appearance of a defaulted place-marker is interpreted as though **PATVARDEFAULT** were ←. Thus the above pattern could have been written as **(match X with (1 =(ADD1 1)))**. Subsequent appearances of a place-marker are interpreted as though **PATVARDEFAULT** were **=**. For example, **(match X with (#1 #1 --))** is equivalent to **(match X with (#1←$1 = #1 --))**, and translates as **(AND (CDR X) (EQUAL (CAR X) (CADR X))**. (Note that **(EQUAL (CAR X) (CADR X))** would incorrectly match with **(NIL)**.)

12.8.6 Replacements

The construct *PATTERN-ELEMENT←FORM* specifies that if the match succeeds, the part of the data that matched is to be *replaced* with the value of *FORM*. For example, if **X =(A B C D E)**, **(match X with ($ 'C $1←Y $1))** will replace the third element of **X** with the value of **Y**. As with assignments, replacements are not performed until after it is determined that the entire match will be successful.

Replacements involving segments splice the corresponding structure into the list being matched, e.g., if **X** is **(A B C D E F)** and **FOO** is **(1 2 3)**, after the pattern **('A $←FOO 'D $)** is matched with

X, **X** will be **(A 1 2 3 D E F)**, and **FOO** will be **EQ** to **CDR** of **X**, i.e., **(1 2 3 D E F)**.

Note that **($ FOO←FIE $)** is ambiguous, since it is not clear whether **FOO** or **FIE** is the pattern element, i.e., whether **←** specifies assignment or replacement. For example, if **PATVARDEFAULT** is **=**, this pattern can be interpreted as **($ FOO← = FIE $)**, meaning search for the value of **FIE**, and if found set **FOO** to it, or **($ = FOO←FIE $)** meaning search for the value of **FOO**, and if found, store the value of **FIE** into the corresponding position. In such cases, the user should disambiguate by not using the **PATVARDEFAULT** option, i.e., by specifying **'** or **=**.

Note: Replacements are normally done with **RPLACA** or **RPLACD**. The user can specify that **/RPLACA** and **/RPLACD** should be used, or **FRPLACA** and **FRPLACD**, by means of CLISP declarations (see page 21.12).

12.8.7 Reconstruction

The user can specify a value for a pattern match operation other than what is returned by the match by writing **(match** *FORM*$_1$ **with** *PATTERN* **= >** *FORM*$_2$**)**. For example, **(match X with (FOO←$ 'A --) = > (REVERSE FOO))** translates as:

```
[PROG ($$2)
  (RETURN
    (COND ((SETQ $$2 (MEMB 'A X))
        (SETQ FOO (LDIFF X $2))
        (REVERSE FOO]
```

Place-markers in the pattern can be referred to from within *FORM*, e.g., the above could also have been written as **(match X with (!#1 'A --) = > (REVERSE #1))**. If **->** is used in place of **= >**, the expression being matched is also *physically changed* to the value of *FORM*. For example, **(match X with (#1 'A !#2) -> (CONS #1 #2))** would remove the second element from **X**, if it were equal to **A**.

In general, **(match** *FORM*$_1$ **with** *PATTERN* **->** *FORM*$_2$**)** is translated so as to compute *FORM*$_2$ if the match is successful, and then smash its value into the first node of *FORM*$_1$. However, whenever possible, the translation does not actually require *FORM*$_2$ to be computed in its entirety, but instead the pattern match compiler uses *FORM*$_2$ as an indication of what should be done to *FORM*$_1$. For example, **(match X with (#1 'A !#2) -> (CONS #1 #2))** translates as **(AND (EQ (CADR X) 'A) (RPLACD X (CDDR X)))**.

12.8.8 Examples

Example: **(match X with (-- 'A --))**

-- matches any arbitrary segment. **'A** matches only an **A**, and the second **--** again matches an arbitrary segment; thus this translates to **(MEMB 'A X)**.

Example: **(match X with (-- 'A))**

Again, **--** matches an arbitrary segment; however, since there is no **--** after the **'A**, **A** must be the last element of **X**. Thus this translates to: **(EQ (CAR (LAST X)) 'A)**.

Example: **(match X with ('A 'B -- 'C $3 --))**

CAR of **X** must be **A**, and **CADR** must be **B**, and there must be at least three elements after the first **C**, so the translation is:

```
(AND (EQ (CAR X) 'A)
   (EQ (CADR X) 'B)
   (CDDDR (MEMB 'C (CDDR X))))
```

Example: **(match X with (('A 'B) 'C Y←$1 $))**

Since **('A 'B)** does not end in **$** or **--**, **(CDDAR X)** must be **NIL**. The translation is:

```
(COND
  ((AND (EQ (CAAR X) 'A)
      (EQ (CADAR X) 'B)
      (NULL (CDDAR X))
      (EQ (CADR X) 'C)
      (CDDR X))
   (SETQ Y (CADDR X))
   T))
```

Example: **(match X with (#1 'A $ 'B 'C #1 $))**

#1 is implicitly assigned to the first element in the list. The **$** searches for the first **B** following **A**. This **B** must be followed by a **C**, and the **C** by an expression equal to the first element. The translation is:

```
[PROG ($$2)
  (RETURN
    (AND (EQ (CADR X) 'A)
       (EQ [CADR (SETQ $$2 (MEMB 'B (CDDR X] 'C)
       (CDDR $$2)
       (EQUAL (CADDR $$2) (CAR X]
```

Example: **(match X with (#1 'A -- 'B 'C #1 $))**

Similar to the pattern above, except that **--** specifies a search for *any* **B** followed by a **C** followed by the first element, so the translation is:

```
[AND (EQ (CADR X) 'A)
   (SOME (CDDR X)
```

```
(FUNCTION (LAMBDA ($$2 $$1)
  (AND (EQ $$2 'B)
    (EQ (CADR $$1) 'C)
    (CDDR $$1)
    (EQUAL (CADDR $$1) (CAR X]
```

D

O

P

Q

R

?

@

[This page intentionally left blank]