THE EXPLORER™ SYSTEM SOFTWARE MANUALS

Mastering the Explorer Environment	Explorer Technical Summary	2243189-0001
	Introduction to the Explorer System	2243190-0001
	Explorer Zmacs Editor Tutorial	2243191-0001
	Explorer Glossary	2243134-0001
	Explorer Networking Reference	2243206-0001
	Explorer Diagnostics	2533554-0001
	Explorer Master Index to Software Manuals	2243198-0001
	Explorer System Software Installation Guide	2243205-0001
Programming With the Explorer	Explorer Programming Concepts	2549830-0001
	Explorer Lisp Reference	2243201-0001
	Explorer Input/Output Reference	2549281-0001
	Explorer Zmacs Editor Reference	2243192-0001
	Explorer Tools and Utilities	2549831-0001
	Explorer Window System Reference	2243200-0001
Explorer Options	Explorer Natural Language Menu System User's Guide	2243202-0001
	Explorer Relational Table Management System User's Guide	2243203-0001
	Explorer Grasper User's Guide	2243135-0001
	Explorer Prolog User's Guide	2537248-0001
	Programming in Prolog, by Clocksin and Mellish	2537157-0001
	Explorer Color Graphics User's Guide	2537157-0001
	Explorer TCP/IP User's Guide	2537150-0001
	Explorer LX™ User's Guide	2537225-0001
	Explorer LX System Installation	2537227-0001
	Explorer NFS™ User's Guide	2546890-0001
	Explorer DECnet™ User's Guide	2537223-0001
	Personal Consultant™ Plus Explorer	2537259-0001
System Software Internals	Explorer System Software Design Notes	2243208-0001
	Release Information, Explorer System Software	2549844-0001

THE EXPLORER™ SYSTEM HARDWARE MANUALS

System Level Publications	Explorer 7-Slot System Installation	2243140-0001
	Explorer System Field Maintenance	2243141-0001
	Explorer System Field Maintenance Documentation Kit	2243222-0001
	Explorer System Field Maintenance Supplement	2537183-0001
	Explorer System Field Maintenance Supplement Documentation Kit	2549278-0001
	Explorer NuBus™ System Architecture General Description	2537171-0001
System Enclosure Equipment Publications	Explorer 7-Slot System Enclosure General Description	2243143-0001
	Explorer Memory General Description (8-megabytes)	2533592-0001
	Explorer 32-Megabyte Memory General Description	2537185-0001
	Explorer Processor General Description	2243144-0001
	68020-Based Processor General Description	2537240-0001
	Explorer II Processor and Auxiliary Processor Options General Description	2537187-0001
	Explorer System Interface General Description	2243145-0001
	Explorer NuBus Peripheral Interface General Description (NUPI board)	2243146-0001
Display Terminal Publications	Explorer Display Unit General Description	2243151-0001
	CRT Data Display Service Manual, Panasonic code number FTD85055057C	2537139-0001
	Model 924 Video Display Terminal User's Guide	2544365-0001
143-Megabyte Disk/Tape Enclosure Publications	Explorer Mass Storage Enclosure General Description	2243148-0001
	Explorer Winchester Disk Formatter (ADAPTEC) Supplement to Explorer Mass Storage Enclosure General Description	2243149-0001
	Explorer Winchester Disk Drive (Maxtor) Supplement to Explorer Mass Storage Enclosure General Description	2243150-0001
	Explorer Cartridge Tape Drive (Cipher) Supplement to Explorer Mass Storage Enclosure General Description	2243166-0001
	Explorer Cable Interconnect Board (2236120-0001) Supplement to Explorer Mass Storage Enclosure General Description	2243177-0001
143-Megabyte Disk Drive Vendor Publications	XT-1000 Service Manual, 5 1/4-inch Fixed Disk Drive, Maxtor Corporation, part number 20005 (5 1/4-inch Winchester disk drive, 112 megabytes)	2249999-0001
	ACB-5500 Winchester Disk Controller User's Manual, Adaptec, Inc., (formatter for the 5 1/4-inch Winchester disk drive)	2249933-0001

1/4-Inch Tape Drive Vendor Publications	Series 540 Cartridge Tape Drive Product Description, Cipher Data Products, Inc., Bulletin Number 01-311-0284-1K (1/4-inch tape drive)	2249997-0001
	MT01 Tape Controller Technical Manual, Emulex Corporation, part number MT0151001 (formatter for the 1/4-inch tape drive)	2243182-0001
182-Megabyte Disk/Tape Enclosure MSU II Publications	Mass Storage Unit (MSU II) General Description	2537197-0001
182-Megabyte Disk Drive Vendor Publications	Control Data® WREN™ III Disk Drive OEM Manual, part number 77738216, Magnetic Peripherals, Inc., a Control Data Company	2546867-0001
515-Megabyte Mass Storage Subsystem Publications	SMD/515-Megabyte Mass Storage Subsystem General Description (includes SMD/SCSI controller and 515-megabyte disk drive enclosure)	2537244-0001
515-Megabyte Disk Drive Vendor Publications	515-Megabyte Disk Drive Documentation Master Kit (Volumes 1, 2, and 3), Control Data Corporation	2246129-0002
	Volume 1, General Description, Operation, Installation and Checkout, and Part Data	2246125-0004
	Volume 2, Theory, General Maintenance, Trouble Analysis, Electrical Checks, and Repair Information	2246125-0005
	Volume 3, Diagrams	2246125-0006
1/2-Inch Tape Drive Publications	MT3201 1/2-Inch Tape Drive General Description	2537246-0001
1/2-Inch Tape Drive Vendor Publications	Cipher CacheTape® Documentation Manual Kit (Volumes 1 and 2 With SCSI Addendum and, Logic Diagram), Cipher Data products	2246130-0001
	1/2-Inch Tape Drive Operation and Maintenance (Volume 1), Cipher Data Products	2246126-0001
	1/2-Inch Tape Drive Theory of Operation (Volume 2), Cipher Data Products	2246126-0002
	SCSI Addendum With Logic Diagram, Cipher Data Products	2246126-0003

Printer Publications

Model 810 Printer Installation and Operation Manual	2311356-9701
Omni 800™ Electronic Data Terminals Maintenance Manual for Model 810 Printers	0994386-9701
Model 850 RO Printer User's Manual	2219890-0001
Model 850 RO Printer Maintenance Manual	2219896-0001
Model 850 XL Printer User's Manual	2243250-0001
Model 850 XL Printer Quick Reference Guide	2243249-0001
Model 855 Printer Operator's Manual	2225911-0001
Model 855 Printer Technical Reference Manual	2232822-0001
Model 855 Printer Maintenance Manual	2225914-0001
Model 860 XL Printer User's Manual	2239401-0001
Model 860 XL Printer Maintenance Manual	2239427-0001
Model 860 Xl Printer Quick Reference Guide	2239402-0001
Model 860/859 Printer Technical Reference Manual	2239407-0001
Model 865 Printer Operator's Manual	2239405-0001
Model 865 Printer Maintenance Manual	2239428-0001
Model 880 Printer User's Manual	2222627-0001
Model 880 Printer Maintenance Manual	2222628-0001
OmniLaser™ 2015 Page Printer Operator's Manual	2539178-0001
OmniLaser 2015 Page Printer Technical Reference	2539179-0001
OmniLaser 2015 Page Printer Maintenance Manual	2539180-0001
OmniLaser 2108 Page Printer Operator's Manual	2539348-0001
OmniLaser 2108 Page Printer Technical Reference	2539349-0001
OmniLaser 2108 Page Printer Maintenance Manual	2539350-0001
OmniLaser 2115 Page Printer Operator's Manual	2539344-0001
OmniLaser 2115 Page Printer Technical Reference	2539345-0001
OmniLaser 2115 Page Printer Maintenance Manual	2539356-0001

Communications Publications

990 Family Communications Systems Field Reference	2276579-9701
EI990 Ethernet® Interface Installation and Operation	2234392-9701
Explorer NuBus Ethernet Controller General Description	2243161-0001
Communications Carrier Board and Options General Description	2537242-0001

Omni 800 is a trademark of Texas Instruments Incorporated.

OmniLaser is a trademark of Texas Instruments Incorporated.

Ethernet is a registered trademark of Xerox Corporation.

CONTENTS

4 Characters

5 Packages

Paragraph **Title** **Page**

6 Lists and List Structure

7 Arrays

8 Strings

9 Sequences

10 Structures

11 Hash Tables

12 Type Specifiers

13 Declarations

14 Control Structures

15 Loop Iteration Macro

16 Functions

17 Closures

18 Macros

22 The Disassembler

23 Maintaining Large Systems

27 Processes

28 Initializations

29 Locatives

A Zitalisp Compatibility

Index

Figures

Tables

ABOUT THIS MANUAL

Purpose

The purpose of this manual is to provide reference information dealing with the concepts and functional descriptions of the core Lisp language. Common Lisp, as defined in Guy Steele Jr.'s *Common LISP: The Language* (Burlington, MA: DEC Press, 1984), is the primary dialect of Lisp used in the Explorer system. However, Explorer Lisp also includes a strong Zetalisp heritage, as defined in the *MIT Lisp Machine Manual*, 6th edition (also known as the Orange book).

Common Lisp and the Explorer System

Common Lisp is the newly evolving standard for the Lisp language. As the definition of the language becomes more comprehensive, Texas Instruments will continue to conform to the standard. At this time, however, there are two major concerns. First, the present definition of Common Lisp does not cover the full functionality of the current Explorer system. For example, Common Lisp does not define a programming construct comparable to the Massachusetts Institute of Technology (MIT) flavor system. This omission is not an oversight; a Common Lisp object-oriented programming system has simply not yet been specified. Because the flavor system does not currently conflict with any Common Lisp functionality, the definition of flavors is included as an extension to the Explorer implementation of Lisp. In general, this same line of reasoning was used for incorporating other Zetalisp functionality.

The second major concern is that some Zetalisp functions have been redefined by Common Lisp while others have been made obsolete because a better solution has been made available. To support those programs written in Zetalisp, the Explorer also supports the Zetalisp dialect. This manual, however, documents only Common Lisp functions and the extensions mentioned previously. Zetalisp functions deemed obsolete or superseded appear in Appendix A with a brief description or a reference to their Common Lisp equivalents.

Other Manuals

While most Common Lisp functions and core extensions are documented in this book, several other Explorer manuals document functions used in the system. In particular, the *Explorer Input/Output Reference* manual is a companion to this manual, documenting Common Lisp input/output (I/O) functions as well as those that are Explorer extensions. The *Introduction to the Explorer System* and *Explorer Tools and Utilities* manuals discuss in more detail many of the basic concepts of the Lisp machine environment and the Explorer system in particular. The *Explorer Window System Reference* manual describes the complete functionality of the window system, and the *Explorer Networking Reference* manual describes many functions that are implementation dependent for the Explorer.

1

INTRODUCTION

How to Read This Manual

1.1 Although you can read the main body of this book in any order, you should read Section 1 first. It contains information that is useful and necessary for a complete understanding of the reference material contained in the rest of the manual.

Notational Conventions

1.2 The functional description of a Lisp form starts with a syntax line, which includes the following:

1. The name of the particular form being described is the first object of the syntax line. It is always in the **boldface** font.
2. All arguments of the form follow the name. They are in the *italic* font. (When the declaration **ignore** is provided in the syntax line, a value must be provided in the call form, but this value is never used.)
3. All lambda-list keywords—&optional, &rest, &key, and so on—are in the medium font and always follow the required arguments.
4. All keyword arguments are preceded by a colon and appear in the **boldface** font.
5. The form type indicator is used to identify the particular type of the form. It is located in the far right-hand corner of the syntax line. The different form types discussed in this manual are functions, special forms, macros, variables, constants, flavors, methods, and type specifiers.
6. Some syntax lines include a status indicator just to the left of the form type indicator. The status indicator specifies whether the form is Common Lisp (*[c]*) or an Explorer extension (no status indicator).
7. Descriptions of related forms are presented together and are discussed within the same paragraphs. In this case, the syntax lines for the forms are stacked (one on top of another).

The following are examples of different syntax lines.

yo-function *argumentA argumentB* &optional *argumentC* *[c]* Function
&key **:keyword1 :keyword2 :keyword3**

This syntax line has two required arguments (*argumentA* and *argument B*), a lambda-list keyword (&optional), an optional argument (*argumentC*), another lambda-list keyword (&key), and three optional keyword arguments (**:keyword1**, **:keyword2**, and **:keyword3**). It is also too long to fit on one line, thus the continuation line. Note also the status indicator (*[c]*) showing that this function is standard Common Lisp.

related-function1 *arg1 arg2* Function
related-function2 *arg1 arg2* Function
related-function3 *arg1 arg2* Function
related-function4 *arg1 arg2* Function

This is an example of four related functions; note the stacking of the syntax lines to emphasize the relationship between the functions.

Syntax Line for Special Forms and Macros

1.2.1 The syntax line for a special form or a macro is more complicated than the previously discussed syntax lines because special forms and macros often have subforms instead of, or in addition to, arguments. To notate the syntax line of special forms and macros, the following conventions have been established:

- Brackets [ ] used within the syntax line of a macro or special form indicate an optional subform or argument.

- Braces { } used within the syntax line of a macro or special form simply indicate a grouping of subforms.

  - Braces followed by an asterisk (*) are used to indicate zero or more repetitions of the subforms.

  - Braces followed by a plus sign (+) indicate one or more repetitions of the subforms.

- A vertical bar | can appear within a set of brackets or braces to indicate a choice between either of the objects separated by the bar.

- The dot notation (.) indicates that the subform following the dot represents a list of subforms, not just a single subform. The dot can only appear just prior to the last subform of the syntax line. It normally is used to set off the body of a form.

The following are some examples of the syntax line of special forms and macros.

a-special-form *arg* [*form*] Special Form

In this syntax line, there is one argument and one optional subform.

a-macro *arg* ({*var*|*val*}*) Macro

In this syntax line, there is one required argument and a subform list that can contain zero or more subforms. Each subform of the list consists of either *var* or *val*.

another-macro *arg* ({*var*|*val*}+) Macro

This syntax line is like the previous one, but the subform list must contain at least one subform.

another-sp ({*form1*}+) [*form2*] Special Form

This syntax line has a required subform list but can also accept an additional optional form.

one-more-sp ({*form1*}+) . [*form2*] Special Form

This syntax line is like the previous syntax line, but the dot notation preceding *form2* means that it should be a list of forms rather than just a single form (an atom here would create a dotted list).

Example Conventions

1.2.2 The examples in this manual are to be interpreted as if they were typed into the Lisp Listener and evaluated. When an evaluated object returns a value, this return is indicated with the => characters. Similarly, macro expansion is indicated with the ==> characters. For example:

```
(+ 1 2) => 3

(for i 1 5 (print (* i i)))
==>(do ((i 1 (+ i 1)))
       ((> i 5))
     (print (* i i)))
```

Equivalence is indicated with the <=> characters. For example, the following forms are equivalent:

```
(form a) <=> (form b)
```

Use of Typefaces

1.2.3 Three fonts are used in this manual to denote Lisp code:

- System-defined words and symbols are in **boldface**. System-defined words and symbols include names of functions, variables, macros, flavors, methods, keywords, and so on.
- Examples of programs and output are in a special `monowidth` font.
- Sample names are in *italics*. Names in italics are placeholders for any value you choose to substitute. Thus, arguments in a description are in italics. Italics are also used for emphasis and to introduce new terms.

For example, the following sentence contains the word **setq** in boldface because **setq** is defined by the system:

> The purpose of the **setq** special form is to assign a value to a variable.

Within each example of actual Lisp code, all names are shown in the monowidth font. For example:

```
(setq x 1 y 2) => 2
(+ x y) => 3
```

> The form `(setq x 1 y 2)` sets the variables `x` and `y` to integer values; then, the form `(+ x y)` adds them together.

In this example, `setq` appears in the monowidth font because it is part of a specific example.

Occasionally, in examples where you could substitute a specific value, the boldface and italic fonts are used together.

Lisp Modes

1.3 Two dialects of Lisp are supported on the Explorer system: Common Lisp, which is the primary dialect, and Zetalisp. Because the two dialects have some inherent incompatibilities, your Lisp process must execute in either Common Lisp mode or Zetalisp mode—it cannot execute in both at the same time. The main differences involve the way in which **read**, **print**, **eval**, and **compile** work. It is possible for functions defined in one mode to call functions defined in the other. The body of this manual assumes Common Lisp mode; refer to Appendix A for the Zetalisp differences.

The difference between Zetalisp and Common Lisp mode centers around the accessibility of two kinds of symbols: those that define functions considered obsolete in a Common Lisp environment and those whose names have meanings incompatible between the two modes.

All of the symbols associated with Zetalisp mode are supplied as part of a Zetalisp-Compatibility system. In Release 3, this system is loaded and accessible. However, in subsequent releases, the Zetalisp-Compatibility system will have to be loaded into the environment by the user. Although it will be available in source form as part of the Explorer product, its functionality will not be extended.

Mode Implementation

1.3.1 All of the obsolete and incompatible symbols in Zetalisp mode are defined in a package called ZLC. The ZLC package has an alias called GLOBAL that is functionally compatible with the Zetalisp definition. The USER package inherits access to all of the obsolete symbols regardless of the mode you are using. The incompatible symbols are defined as internal in ZLC, so they are not normally visible. When you switch to Zetalisp mode, a flag is set, which tells the Lisp Reader to access the internal/incompatible Zetalisp symbols instead of the ones defined in the LISP package. The evaluator must also look at this flag because special variables are also handled differently in Zetalisp mode. See Section 5, Packages, for complete details on packages.

You can set the variable **compiler:*warn-of-superseded-functions-p*** to true to cause the compiler to issue warnings about the use of functions in the ZLC package.

The number of Common Lisp and Zetalisp symbols that conflict in this way is very small. Thus, most of the Zetalisp primitives are available in both modes and are referred to in this book as Explorer extensions to Common Lisp. In the future, as the Common Lisp definition expands, conflicts that arise due to these extentions will be resolved by isolating these Zetalisp symbols in Zetalisp mode.

Using the Two Modes on the Explorer System

1.3.2 The Explorer system normally runs in Common Lisp mode; however, under certain circumstances, the mode is switched to Zetalisp. The function **lisp-mode** returns the value **:common-lisp** or **:zetalisp**, depending on which mode is currently in effect.

You can change modes with the function **set-lisp-mode**. However, if you only want one form to be read in the alternate mode, you can use the reader macros **#!Z** or **#!C** to read the form in Zetalisp mode or Common Lisp mode, respectively. For example:

```
#!Z (member 2 '(1 2 3)) => (2 3)

#!C (member 2 '(1 2 3) :test #'equal) => (2 3)
```

Once the form has been read, the mode reverts back to whatever it was prior to the reader macro. Note that this example demonstrates the use of two different functions: the first of the preceding forms invokes the `zlc:member` function, whereas the second form invokes the `lisp:member` function. However, this does not mean that a function is evaluated in an alternate mode if it is not defined in that mode. For example, the form `#!Z(find-symbol "EVAL")` finds the **eval** symbol in the LISP package where it is defined rather than in the ZLC package.

lisp-mode Function

This function returns the keyword symbol indicating which mode is currently in effect.

set-lisp-mode *keyword* &optional *globally-p* Function

This function switches the current Lisp mode to what is specified by the value of *keyword*, which should be either **:common-lisp** or **:zetalisp**. If *globally-p* is unspecified or is specified as **nil**, then the mode is switched only for the environment in which it is executed. If *globally-p* is specified as true, then the mode is switched for the global environment.

turn-common-lisp-on Function

This function sets the Lisp evaluation mode for the global environment to Common Lisp; it does nothing if the mode is already Common Lisp.

turn-zetalisp-on Function

This function sets the Lisp evaluation mode for the global environment to Zetalisp; it does nothing if the mode is already Zetalisp.

Using the Two Modes From Zmacs

1.3.3 When you are editing Lisp code from the Zmacs editor, be sure to note the mode in which you are editing. In this context, *mode* refers to the major editing mode of Zmacs for a particular buffer. The editing mode is displayed in the mode line near the bottom of the window at the top of the Zmacs minibuffer. Zmacs needs to know if the file you are editing is Common Lisp or Zetalisp because the interpretation of certain characters changes the syntactic parsing of Lisp forms. If you press the BREAK key, the Listener provided for you in the typeout window is in the same mode as that for the buffer you were editing.

When buffers are created for editing Lisp code, you should determine which editing mode you want to use. Use the Zmacs extended commands Common Lisp Mode or Zetalisp Mode to choose the appropriate mode. When you execute either of these commands, you should answer `Yes` to the prompt asking if you want to insert the file attribute line as the first line of your buffer. For example, when a buffer is using Common Lisp mode, the file attribute line appears as follows:

```
;;;           -*- Mode:Common-Lisp -*-
```

When a buffer is using Zetalisp mode, the file attribute line appears as follows:

```
;;;           -*- Mode:Zetalisp -*-
```

NOTE: File attribute lines in older Zetalisp programs declare their mode to be `Lisp`. Thus, for purposes of compatibility, the attribute `Mode:Lisp` is translated by the Explorer to mean `Mode:Zetalisp` unless it is accompanied by the attribute `Syntax:Common-Lisp` or `Readtable:Common-Lisp`. (These attributes are used by other MIT-derived Lisp machines and are equivalent to `Mode:Common-Lisp`.)

2

SYMBOLS

Symbol Definitions

2.1 A *symbol* is a Lisp data object that defines a relationship between a print name, a package, a property list, a function definition, and a value, each of which is stored in its own cell.

The *print name* cell contains a string used to identify the symbol. Once a print name is established for a symbol, you are not allowed to change it.

The *package* cell must contain a package object or the value **nil**. If this cell refers to a package, the symbol is said to be *owned* by that package.

The *property list* cell allows you to associate supplemental attributes with a symbol. Sometimes called *plists*, property lists are lists of alternating names and values. Several functions are provided for manipulating property lists. Initially, the property list cell for a symbol is **nil**.

The *function definition* cell contains a function object. When a symbol name is evaluated in the context of a function call, this is the definition that is used. If no function object has been assigned for the symbol, then an undefined function error is generated.

The *value* cell is used to refer to a Lisp object. When the symbol is evaluated as a special variable, this is the value that is returned. An *unbound* symbol is one that has not been assigned a data value. Evaluating an unbound symbol produces an error.

Symbols have two fairly distinct uses. An *interned symbol* is one that can be referred to by its print name. The reference is made possible by the package system. An internal symbol is present in a package and is defined to have a unique print name among all other accessible symbols, including inherited and imported symbols. When the Lisp Reader attempts to parse a symbol name, it tries to match that name with one of the associated print names in the current namespace. You can switch the namespace context by explicitly supplying a package name to the Reader (which defines a different namespace). When interned symbols are printed, the print name is used; the print name is preceded by the package name if the symbol is not accessible in the current namespace (see Section 5, Packages, for more details).

An *uninterned symbol* is one that is not in any package and, therefore, cannot be referred to by its print name. Nevertheless, uninterned symbols are still useful objects. References to them can be maintained in other ways, such as in lists or arrays or by using them as data values to other interned symbols. In some applications, these symbols should be inaccessible by name to avoid known print name conflicts. When an uninterned symbol is printed, it is preceded by a #: prefix.

Naming Symbols

2.2 A symbol name can contain both alphabetic and numeric characters. However, all symbol names must contain at least one nonnumeric character that differentiates the symbol name from a number. For the purposes of symbol names, the following ASCII characters are also considered to be alphabetic:

```
+ - * / @ $ % ^ & _ < > ~ . =
```

Note that a period or several periods by themselves are not acceptable symbol names.

Special Characters

2.3 Both uppercase and lowercase characters are acceptable in symbol names; however, the Lisp Reader normally converts lowercase letters to uppercase. You can force the Reader to accept lowercase letters and other usually unacceptable characters by preceding them with a backslash (\). The backslash is a signal to the Reader to parse the next character in a protected manner. If the character is lowercase, it is not converted. If the character is nonalphabetic, it is explicitly treated as alphabetic. A single backslash is not included as part of the name. However, if you include two consecutive backslashes in a symbol name, the second backslash becomes part of the name. You can also protect characters in a symbol name by enclosing them in vertical bars (| |). When the Reader parses a token and detects either the backslash or a pair of vertical bars, it assumes that token must be a symbol name. For example:

```
1+                  ;a symbol
+1                  ;a number
\+1                 ;a symbol
ax^2+bx+c           ;a symbol
TEST                ;a symbol
Test                ;a symbol EQ to TEST
Miles/Hour          ;a symbol
1                   ;a number
\1                  ;a symbol
|1|                 ;a symbol EQ to \1
\\                  ;a symbol
|\\|                ;a symbol EQ to \\
\.                  ;a symbol
\|                  ;a symbol
|\||                ;a symbol EQ to \|
```

Besides the backslash and the vertical bar, the Lisp Reader treats several other characters specially, such as the open parenthesis and the sharp-sign (#). The special meanings of these characters are defined by the Explorer system and are explained in the *Explorer Input/Output Reference* manual. The following characters are special because, though they are normally treated as alphabetic characters, you can use them for defining your own Common Lisp Reader macros:

```
? ! [ ] { }
```

The double meaning of these characters can cause problems if one Common Lisp system uses them to create symbols and another system defines them to have some other special meaning for the Reader.

The Explorer also includes many non-ASCII characters that can be used as alphabetic characters in symbols. For a complete list of the extra non-ASCII characters, see Table 4-1. Note that these characters and their behavior are not defined by the Common Lisp standard; therefore, programs that use them are not likely to be portable to other Common Lisp systems.

References

2.4 In general, the term *reference* denotes a mapping of a name to some other object. Typically, some kind of structure has been created to perform this mapping. For instance, when you make a reference to a symbol name, the object that is mapped to is the data value or, in some contexts, the function definition. In practice, mappings are made to objects other than data values, and objects other than symbols can be used in references. Regardless of who is making reference to what, the mechanics of making a particular reference involves some fairly strict regulations. The terms *scope* and *extent* can be used to describe these regulations.

Scope

2.4.1 A program in Common Lisp, as in any other programming language, is written using names that refer to the variables, functions, and other entities used by the program. Lisp generally uses symbol objects as names for other entities. Some of the associations of names with what they represent are defined in the global environment, while others are defined locally for use in a limited region of the program. The portion of the source code within which a particular name is defined to represent a particular entity is called the *scope* of that definition.

For example, the following form creates a special variable and defines the symbol `count` to be its name:

```
(defvar count)
```

This definition is said to have *indefinite scope* because the name is defined in the global environment for use anywhere.

On the other hand, the following form defines a local variable (assuming that it is not declared special) that is named `index`:

```
(let (index) ...)
```

This definition is said to have *lexical scope* because the name is defined only for the portion of the source program text that constitutes the body of the `let` form. Local variables are also called *lexical variables*.

Lexical shadowing occurs when a local definition hides a previous definition. For example, the following function `demo` contains two local variables that are both named `x`; one is the argument of the `demo` function and the other is defined by the `let` form:

```
(defun demo (x)
  (print x)
  (let ((x 99))
    (print x) ) )
```

Within the body of the `let`, the name `x` represents the variable defined by the `let`. Normally, the scope of an argument name is the entire function body, but in this case, because inner definitions take precedence over outer definitions, the scope of the argument `x` is that portion of the function body outside of the `let` body. For convenience, however, this manual usually simply indicates that a particular entity has lexical scope over a certain region, with the unstated assumption that the scope could be affected by shadowing.

Extent 2.4.2 The term *extent* means the period of time for which a particular entity exists during the execution of a program. For example, a **let** form establishes a variable binding when the form is entered, and disestablishes the binding when the **let** is exited. Thus, in the following example, the local variable `x` exists from the time the `let` is entered until the time it is exited:

```
(defun f1 (a)
  (let ((x (first a))
    (f2 a)
    (print x)))

(defun f2 (b) ...)
```

Note that extent differs from scope in that the variable `a` continues to exist during the execution of `f2` even though `f2` is outside the scope of `x`.

The term *dynamic extent* is used to describe entities that have explicit points of creation and destruction. For example, a special variable is bound when a **let** or other binding form is entered and is unbound when the form is exited. The term *dynamic scope* is sometimes used to describe things such as special variable bindings that have indefinite scope and dynamic extent.

In Common Lisp, however, most entities have what is called *indefinite extent*. This means that the entity exists as long as there is a possibility of accessing it. For example, local variables have indefinite extent—although they are usually disestablished upon exit from the form that created them. Occasionally references to these variables can still occur at a later time. For example:

```
(defun make-remover (item)
  (function (lambda (sequence)
              (remove item sequence))))

(setq nonzero (make-remover 0))
(funcall nonzero '(3 5 0 2 0 4))  =>  (3 5 2 4)
```

Here, the local variable `item` continues to be accessible by the `lambda` expression function even after control has returned from the `make-remover` function. The function object returned by `make-remover` is called a *lexical closure* because it has associated with it (*closes over*) a lexical entity that it can continue to access whenever it needs to. A further discussion of closures is given in Section 17, Closures.

As a further example of these concepts, Figure 2-1 demonstrates four different combinations of scope and extent. In the lexical scope and dynamic extent example, two pointers are generated, both named `x`. The extent of each reference is limited to the time of entry and exit of the corresponding `let` statement. Note that during the evaluation of the inner `let`, `x` is bound to `2`, temporarily superseding the previous reference whose value is `1`. The scope of the outer `let`'s reference to `x` does not include the textual confines of the inner `let`. This is an example of lexical shadowing; the binding of `x` in the inner `let` shadows the reference to `x` in the outer `let`.

The second combination shown in Figure 2-1, lexical scope and indefinite extent, demonstrates a lexical closure created when the `lambda` form is converted to a function. The closure that is returned from `set-number` retains the reference to the value of `x` established in the `lambda`. The extent for this reference is the period of time during which this lexical closure exists. Because `x` was defined as a parameter inside the `defun`, this particular reference to `x` cannot be used textually outside of `set-number`. Thus, the attempt to `setq` the value of `x` to `1` does not alter the value of `x` within `set-number`, which is called when `remember` is `funcalled`.

The third combination, indefinite scope and dynamic extent, demonstrates that some variable `x` can appear anywhere in the text of the program, but the extent of the reference to a data object is limited. Note that in Figure 2-1 `tester` is acceptable because the reference to `x` is proclaimed special. However, in this example, `x` has the value of `1` only within the dynamic extent of the `let` in the `declarer` function. If `x` had a different value at the Lisp top level, then the binding of `x` within the `let` statement would be an example of *dynamic shadowing*.

Figure 2-1

Scope and Extent

Lexical Scope and Dynamic Extent

```
(let ((x 1))
  (list x
    (let ((x 2))
      x)
    x))
=> (1 2 1)
```

Lexical Scope and Indefinite Extent

```
(defun set-number (x)
   (function
     (lambda ( ) x)))

(setq remember (set-number 0))

(funcall remember) => 0

(setq x 1) => 1

(funcall remember) => 0
```

Indefinite Scope and Dynamic Extent

```
(proclaim '(special x))

(defun declarer ( )
   (let ((x 1)
     (tester)))

(defun tester ( )
   x)

(declarer) => 1
```

Indefinite Scope and Indefinite Extent

```
;At the Lisp top level

(defvar x 1)
```

The fourth combination shown in Figure 2-1, indefinite scope and indefinite extent, illustrates that a reference can occur anywhere in the program and that this reference is in effect anywhere in the program. The variable `x` is declared at the top level should be: and, therefore, can be referred to anywhere at any time, subject only to any shadowing that may occur within a form such as a let.

Although the examples in Figure 2-1 demonstrate the concepts of scope and extent as applied specifically to symbols and variables, these concepts also apply to references in general.

Local Variables

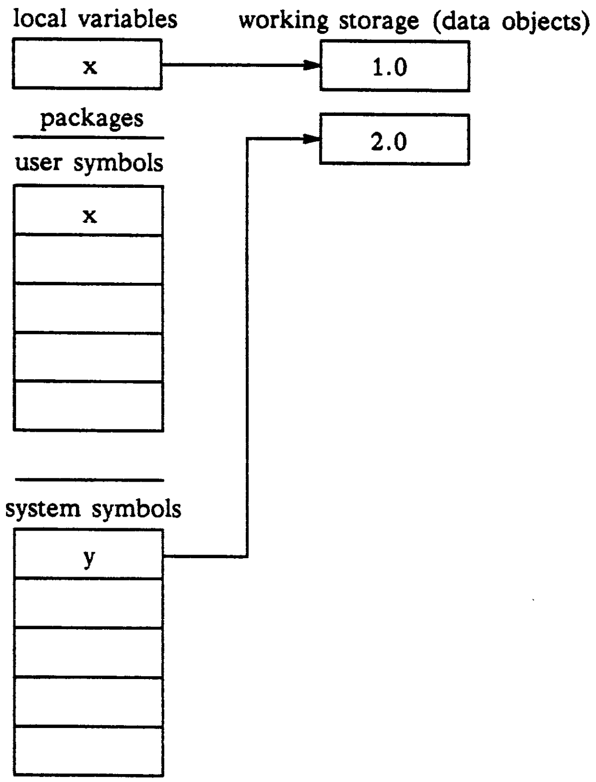
2.5 *Local variable bindings* offer a quick and temporary means of referring to data objects within a limited scope. Local variables are declared in several kinds of Lisp forms, including **lets** and **lambda** expressions. When bindings are made, a table of local variables is created. The value for the local variable is kept in this table. Thus, if this variable is shadowing the scope of another variable, the previous value is maintained, though it cannot be referred to. For example, Figure 2-2 shows that when `x` is set to `1.0`, a pointer to the data object `1.0` is stored in the local variable. Notice that when the special variable `sys:y` is set to `2.0`, the value cell of that symbol points to the data object `2.0`.

Figure 2-2

How Local Variables Operate

```
;; Example code
(proclaim '(special sys:y))

(defun example ( )
   (let ((x nil))
     (setf x 1.0)
     (setf sys:y 2.0)
           . . . ))
```



Creating Symbols

2.6 The creation of interned symbols is described in Section 5, Packages. The following functions are used for creating uninterned symbols.

make-symbol *print-name* — *[c]* Function
make-symbol *print-name* &optional *permanent-p* — Function

This function creates an uninterned symbol whose print name is the string *print-name*. The value and function definition cells remain unbound and the property list remains empty.

The Explorer allows an optional second argument, *permanent-p*, which when specified as true, indicates that the symbol should be in a permanent memory area, and its print name is copied to the proper area. This feature is used by **intern** but is unlikely to be needed elsewhere. If *permanent-p* is **nil** (the default), the print name is placed in the default area. Consider the following example:

```
(setf x (make-symbol "FOO")) => #:FOO

(set x 0) => 0

(eval x) => 0

x => #:FOO
```

Note that in this example, the symbol `x`, when used as an argument to the `set` and `eval` functions, is evaluated before these functions perform their respective operations. Thus, in the `set` form, `a` is evaluated, returning `#:FOO`, which is then set to 0. Similarly, in the `eval` form, `x` is evaluated, returning `#:FOO`, which is then evaluated itself, returning 0 as the result of `eval`.

copy-symbol *symbol* &optional *copy-props-p* — *[c]* Function

This function returns a new uninterned symbol with the same print name as *symbol*. If *copy-props-p* is non-**nil**, then the value of the value and function definition of the new symbol is the same as those of *symbol*, and the property list of the new symbol is a copy of *symbol*'s. If *copy-props-p* is **nil**, then the new symbol's function and value are void and its property list is empty. For example:

```
(setf x (make-symbol "foo")) => foo
symeval x) => ERROR
```

Note that the symbol is *not* interned; it is simply created and returned.

gensym &optional *x* — *[c]* Function

This function creates a print name, as well as a new symbol associated with that print name. The **gensym** function returns the new, uninterned symbol.

The newly created print name contains a prefix (whose default is G), followed by the decimal representation of a number. This number is incremented by 1 for every call to **gensym**.

The *x* argument can specify either the original value of the counter contained in the print name or a string that specifies the default prefix for the print name. If it is an integer, the *x* argument must be nonnegative; otherwise, it is used as a prefix to the **gensym** symbol name. This prefix is retained for all subsequent calls to **gensym** until you change it. Once it has processed this argument, **gensym** generates a symbol, just as it would without an argument. For example:

```
(gensym) =>  #:G4679
(gensym "SYMBA-") =>  #:SYMBA-4680
(gensym 66) =>  #:SYMBA-0066
(gensym) =>  #:SYMBA-0067
(gensym "CHEETAH-") => #:CHEETAH-0068
```

The function **gensym** is usually used to create a symbol that should not normally be seen by the user and whose print name is unimportant, except to allow easy visual distinction between two such symbols. The optional argument is rarely supplied. The name **gensym** comes from the term *generate symbol*, and the symbols produced by it are often called *gensyms*.

If you want a symbol like that produced by **gensym** but also want to intern that symbol, use **gentemp**.

gentemp &optional *prefix package* *[c]* Function

This function generates a symbol whose name begins with *prefix* (which is a string or a symbol name) and interns that symbol in *package* (unlike **gensym**, which produces an uninterned symbol). This function creates the symbol name by concatenating *prefix* with a number, which is incremented each time **gentemp** is called to ensure that each symbol name is unique. The default for *prefix* is **T**, and the default for *package* is the current package. However, unlike **gensym**, **gentemp** does not allow you to reset the internal counter. Furthermore, the *prefix* for **gentemp** is used for only one function call, unlike when you specify a string prefix for the argument to **gensym**.

Value Cell

2.7 The following functions are associated with the value cell for a symbol.

symbol-value *symbol* *[c]* Function

This function is the basic primitive for retrieving a symbol's value. The form (**symbol-value** *symbol*) returns the current dynamic binding of *symbol*. Generally speaking, this is the function called by **eval** when it is given a symbol to evaluate. An error is signaled if this function is applied to an unbound symbol. This function can be used as a *place* form to **setf** (see paragraph 2.13, Generalized Variables).

symeval-globally *symbol* Function

This function returns the value of the global binding of *symbol* even if that symbol is shadowed by a local variable. An error is signaled if the symbol is not bound globally.

makunbound *symbol* *[c]* Function
variable-makunbound *symbol* Special Form

The function **makunbound** removes the current dynamic binding of the special variable named by the argument *symbol*. For example:

```
(setf a 5)
a => 5
(makunbound 'a)
a => ERROR: unbound variable
```

The special form **variable-makunbound** performs the same operation, but it does not evaluate its argument and it can be used on flavor instance variables as well as on special variables.

makunbound-globally *symbol* Function

This function causes the global binding of *symbol* to be become unbound.

set *symbol-form value-form* *[c]* Function

This function is the primitive used for assigning values to symbols. When the argument *symbol-form* is evaluated, the value cell of the returned object of *symbol-form* is assigned to the value of *value-form*. The argument *value-form* is also evaluated, and its value is the returned object of the **set** function. This function sets only current dynamic bindings; do not use **set** for changing the values of local variables (use **setf** instead). For example:

```
a => x
b => x
(set (if (eq a b) 'c 'd) 'foo) => foo
c => foo
```

This form either sets `c` to `foo` or sets `d` to `foo`, depending on the value of the test `(eq a b)`.

set-globally *symbol-form value-form* Function

This function is like the **set** function but sets the global binding of *symbol-form* to *value-form*.

Function Definition Cell

2.8 The following functions are associated with the function definition cell for a symbol.

symbol-function *symbol* *[c]* Function

This function returns the current global function definition of *symbol*, that is, the contents of its function definition cell. An error is signaled if this function is applied to a symbol that does not have a function definition.

Note that the **symbol-function** function operates only on global functions. Thus, it cannot access the values of local functions defined by **flet** or **labels** (see paragraph 16.9, Defining Local Functions).

You can modify the global function definition for a symbol by using the macro **setf** (see paragraph 2.13, Generalized Variables) with a **symbol-function** form as a generalized variable. Once you have made this modification, the old function definition of the symbol is lost. You should use this type of modification only if the functional definition object is a function or a macro, not a special form, because the compiler would still recognize the name of the object as being a special form and would parse the code according to the requirements for the previous definition.

fmakunbound *symbol* *[c]* Function

This function causes the *symbol* to have no function definition. For example:

```
(defun foo (x)
  (+ x 3))
(foo 2) => 5

(fmakunbound 'foo)
(foo 2) => ERROR: undefined function.
```

Print Name

2.9 The following function is associated with the print name for a symbol.

symbol-name *symbol* *[c]* Function

This function returns the print name of the argument *symbol*. For example:

```
(symbol-name 'abc) => "ABC"
(symbol-name :abc) => "ABC"
(symbol-name 'sys:abc) => "ABC"
```

Package Cell

2.10 The following function is used to return information about the package cell for a symbol.

symbol-package *symbol* *[c]* Function

This function returns the contents of the package cell of the argument *symbol*, which must be a symbol. You can use this function as a *place* form to **setf** (see paragraph 2.13, Generalized Variables).

Property List Cell

2.11 The following functions are used to return information about the property list cell for a symbol. For more information about property lists, see Section 6, Lists and List Structure.

get *symbol property-name* &optional *default* *[c]* Function

This function returns the value paired with the name *property-name* from the property list specified by *symbol*. The *property-name* argument must be **eq** to a member of *symbol*'s property list. Otherwise, **get** returns *default* if specified, or **nil** if *default* is unspecified. The *symbol* argument can also be a list or a locative whose cdr stores the properties; such a list or locative is sometimes called a disembodied property list.

For example, suppose that the symbol `sixers` has the following property list:

```
(Charles Barkley Julius Erving Moses Malone
 Maurice Cheeks Andrew Toney)
```

Note the following evaluations:

```
(get 'Sixers 'Charles) => Barkley
(get 'Sixers 'Andrew) => Toney
```

symbol-plist *symbol* *[c]* Function

This function returns the property list of *symbol*.

You cannot use the returned value of **symbol-plist** as an argument to **get** because **get** expects a variable name, not a place. However, you can use the result of **symbol-plist** with **getf** (see paragraph 16.9, Property Lists). Note the following equivalence:

```
(get x y) <=> (getf (symbol-plist x) y)
```

Although you can perform a **setf** of **symbol-plist**, you should be very careful when doing so because you could remove important system information.

putprop *symbol x property-name* Function

This function gives *symbol* a property value of *x* whose property name is *property-name*. If *symbol* already has a value for *property-name*, the new value supersedes the old value. This function is obsolete; in Common Lisp, use the following:

```
(setf (get symbol property-name) x)
```

defprop *symbol x property-name* Special Form

This form is almost the same as **putprop**, but it does not evaluate its arguments. This feature makes it more convenient to type.

remprop *symbol property-name* *[c]* Function

This function removes the property pair named by *property-name* from the property list named by *symbol*, returning a non-**nil** value. If *property-name* is not **eq** to a member of the property list specified by *symbol*, **remprop** returns **nil**. Consider the following example:

```
(symbol-plist 'Sixers)
=> (Sedale Threatt Terry Catledge Charles Barkley
    Julius Erving Moses Malone Maurice Cheeks Andrew Toney)

(remprop 'Sixers 'Andrew) => non-nil

(symbol-plist 'Sixers)
=> (Sedale Threatt Terry Catledge Charles Barkley
    Julius Erving Moses Malone Maurice Cheeks)
```

getl *plist property-name-list* Function

This function is like **get**, except that the second argument is a list of property names. This function searches down *plist* (using **eq** as the test) for any of the names in *property-name-list* until it finds the first property pair whose name is contained in *property-name-list*.

The **getl** function returns the portion of the list *plist* beginning with the first such property that it finds. If none of the property names on *property-name-list* are on the property list, **getl** returns **nil**. Consider the following example:

```
(symbol-plist 'Sixers)
=> (Sedale Threatt Terry Catledge Charles Barkley
    Julius Erving Moses Malone Maurice Cheeks)

(getl 'Sixers '(Julius Moses))
=> (Julius Erving Moses Malone Maurice Cheeks)
```

When more than one of the names in *property-name-list* is present in *plist*, the name returned by **getl** depends on the order of the properties and should not be relied on.

Binding and Setting Variables

2.12 The following functions and special forms are associated with binding and setting variables.

setq {*variable value*}* *[c]* Special Form

This special form is used to set the value of one or more variables. The *variable* arguments are not evaluated, and the *value* arguments are evaluated. This special form operates as follows: the first *value* is evaluated, and the first *variable* is set to the result of the evaluation; then the second *value* is evaluated and the second *variable* is set to the result of this evaluation; and so on until all the variable-value pairs are processed. The **setq** form returns the value of the last variable assignment. If no arguments are passed to **setq**, no assignments take place and **nil** is returned. For example:

```
(setq x (+ 3 2 1)
      y (cons x nil)) => (6)
```

In this example, `x` is set to `6`, `y` is set to `(6)`, and the form returns `(6)`. Note that the assignments are performed in sequence, allowing the variable `y` to use the new value of `x`.

See the **setf** macro in paragraph 2.13, Generalized Variables.

setq-globally {*variable value*}+ Macro

This macro is like **setq** but sets *variable*'s global value cell rather than the current binding. The *variable* argument must be a special variable.

This macro is usually used in a login file to set a variable so that it affects all processes, not just the current one.

psetq {*variable value*}* *[c]* Macro

This macro is like the **setq** special form, except that the variables are set *in parallel*; first, all of the *value* forms are evaluated, and then the corresponding *variables* are set to the resulting values. The returned value of the **psetq** form is **nil**. For example:

```
(setq x 1)
(setq y 2)
(psetq x y y x) => nil
x => 2
y => 1
```

See the **psetf** macro in paragraph 2.13, Generalized Variables.

let ({*var* | (*var* [*form*])}*) {*declaration*}* { *body-form*}* *[c]* Special Form

This special form binds in parallel an arbitrary number of variables to the values of corresponding forms. Any of these variables that previously existed have their old values saved before taking on new values. The **let** special form allows these variables to be manipulated within the *body-forms* using these bindings. Once the execution of the **let** form is complete, the bindings are disestablished and any variable that previously existed is bound to its previous value.

The returned value of the **let** special form is the value of the last form of the *body-forms*. If the body is empty, **nil** is returned.

var — The *var* argument is not evaluated. If *var* previously existed, its old value is saved and *var* is set to the value of *form*. If *var* stands by itself, it is set to **nil**.

form — The *form* argument is evaluated. Its returned value becomes the new value of *var*. The default for this argument is **nil**.

declaration — The optional *declaration* argument specifies declarations in effect locally within the **let** form.

body-form — The *body-form* argument forms are evaluated in sequence within the context of the bindings of the variables.

Consider the following example:

```
(let ((a (+ 3 3))
      (b 'foo)
      (c)
       d)
   ...)
```

Within the body of the preceding `let` form, `a` is bound to 6, `b` is bound to the symbol `foo`, and `c` and `d` are both bound to **nil**.

let* ({*var* | (*var* [*form*])}*) {*declaration*}* {*body-form*}* *[c]* Special Form

This special form is the same as **let**, except that the binding is sequential. Each *var* is bound to the value of its *form* before the next *form* is evaluated. This convention is useful when the computation of a *form* depends on the value of a variable bound in an earlier *form*.

If there are no body forms, then **let*** returns **nil**.

If, you do not specify an initial value for some variable *var* in a **let***, then it is initialized to **nil**. However, you should leave such a variable un-initialized only if you plan to later store a value in it (for example, with **setf**). If you actually want *var* set to **nil**, then it is clearer to specify (*var* **nil**).

let-if *predicate-form* ({*var* | (*var* [*form*])}*) {*body-form*}* Special Form

This special form is a variant of **let** in which the binding of variables is conditional. The variables must all be special variables. If *predicate-form* evaluates to true, then all variables in the variable list are bound to local values. If *predicate-form* is **nil**, then none of the variables in the variable list are bound, and the value forms are not evaluated. The *body-forms* are evaluated regardless of the results of *predicate-form*. The returned value is the value of the last form in the body.

let-globally ({*var* | (*var* [*form*])}*) {*body-form*}* Macro
let-globally-if *predicate-form* ({*var* | (*var* [*form*])}*) {*body-form*}* Macro

This macro is similar to **let**, except that it does not *bind* the *var* variables. Instead, it saves the old values of the variables and then *sets* the variables. The **let-globally** macro then establishes an **unwind-protect** form to reset them to their saved values (see paragraph 14.5, Dynamic Nonlocal Exits). The critical difference is that when the current stack group calls another stack group, the old values of the variables are not restored (see Section 26, Stack Groups). Thus, **let-globally** makes the new values visible in all stack groups and processes that do not bind the variables themselves, not just the current stack group.

The **let-globally-if** macro modifies and restores the variables only if the value of *predicate-form* evaluates to true. The *body-form* is executed in any case.

progv *symbols values {body-form}** *[c]* Special Form

This special form provides extra control over binding. It binds a list of variables dynamically to a list of values and then evaluates the body of the **progv**. The lists of variables and values are computed quantities; this feature is what makes **progv** different from **let**, **prog**, and **do**.

The **progv** form first evaluates *symbols* and *values* and then binds each symbol to the corresponding value. If too few *values* are supplied, the remaining *symbols* are bound to **nil**. If too many *values* are supplied, the excess *values* are ignored.

After the *symbols* have been bound to the *values*, the *body-forms* are evaluated, and finally the symbol bindings are undone. The result returned is the value of the last form in the body. For example:

```
(setf vars '(a b c)
      vals '(1 2 3))
(progv vars vals
   (+ a b c)) => 6
```

progw *vars-vals-form {body-form}** Special Form

This special form is like a **progv** but differs in that the value of the *vars-vals-form*, which is evaluated, should be a list that looks like the first subform of a **let**: ((*var form*)...). Each element of this list is then processed by binding *var* to the value of *form*, just like **let***. Finally, the forms of *body* are evaluated in sequence, returning the value of the last form, and all bindings are undone.

This is a very unusual special form because of the way the evaluator is again called on the result of evaluating *vars-vals-form*. Thus, **progw** is mainly useful for implementing special forms and for functions that call the evaluator.

compiler-let ({*var* | (*var* [*form*])}*) {*body-form*}* *[c]* Special Form

When interpreting Lisp code, this special form behaves like **let** except that each of the variables is implicitly declared special. When a **compiler-let** form is compiled, no code is generated to set up the bindings. Instead, the bindings are established within the compiler environment while the compiler generates the object code for the *body-forms*.

The **compiler-let** function has two uses. First, it is used to notify the compiler of certain situations that should be considered while generating object code for *body-forms*. For example, the following shows the use of **compiler-let** when generating a patch:

```
(compiler-let ((package (find-package "USER"))
               (sys:lisp-mode :common-lisp)
               (*readtable* sys:standard-readtable)
               (sys:*reader-symbol-substitutions* nil))

 (defun foo () ...)     ;patch to redefine the function foo
)
```

The second use of **compiler-let** is to establish environmental flags to communicate to macros how to process their expansions. This operation assumes that the macro expansion depends upon some special variable to tell it how to expand. For example:

```
(defmacro show-value (value)
   (if verbose-on-p `(format t "The value is ~A" ,value)
                    `(print ,value)))
```

Note that the `show-value` macro will expand to either the `format` form or the `print` form, depending on what the value of the special variable `verbose-on-p` is when the expansion takes place.

Consider the following form:

```
(compiler-let ((verbose-on-p t))
   (show-value thing))
```

When interpreted, `verbose-on-p` is set to `t` and declared special so that `show-value` will generate the macro expansion, which will be the `format` statement. The interpreter then evaluates the expansion. During compilation, the `verbose-on-p` is set to `t`, the `show-value` macro is expanded, and then the compiler generates object code for that expansion. In short, the `compiler-let` and the macro call are optimized simply to the `format` form.

As an optimization in interpreted mode, this use of **compiler-let** makes it possible to remember what a macro expanded to so that the next time the code executes it need not be reexpanded. For more information on this topic, read about the function **sys:displaced** in paragraph 18.5, Displacing Macro Calls.

Generalized Variables

2.13 In Common Lisp, a *variable* is an object that remembers one piece of data. The primary operations on a variable are retrieving and changing that piece of data. These operations are sometimes called the *access* and *update* operations. The concept of variables named by symbols can be generalized as any storage location that remembers one piece of data, no matter how that location is named.

Each kind of generalized variable typically has three functions that implement the access, update, and location operations. For example, **car** accesses the car of a cons, **rplaca** updates the car, and **car-location** returns the location of the car.

Rather than having to remember three separate functions for each kind of generalized variable, you can think of the access function as a name for the storage location. Thus, `(aref x 105)` is a name for the 105th element of the array `x`. Rather than having to remember the update function associated with each access function, you can adopt a uniform way of updating storage locations by using the **setf** macro. Similarly, the location of the generalized variable can be obtained by using **locf** (see paragraph 29.2, Functions That Return Locatives).

The following are the forms associated with updating generalized variables.

setf {*place form*}* *[c]* Macro

This macro assigns the value produced by evaluating *form* to the location specified by *place*. The *place* argument must be a variable or one of the forms discussed in the argument description below. If more than one assignment is specified, the assignments are processed sequentially. The returned value is the value of the last assignment or **nil** if no arguments are provided.

The **setf** macro preserves the usual left-to-right order in which the various subforms are evaluated.

Because **setf** returns the result of evaluating the last *form*, the form `(setf (car x) y)` is not equivalent to the form `(rplaca x y)` because `rplaca` returns the entire list `x` with `y` as its new car rather than the result of evaluating `y`. Thus, the equivalent of using **rplaca** to produce the same result and the same returned value as `(setf (car x) y)` would be the following:

```
(let ((var1 x) (var2 y))
  (rplaca var1 var2) var2)
```

The *place* argument is not evaluated but is used to determine how the value of *form* is to be assigned. The *place* argument must be one of the following:

- A variable name which can be a lexical, dynamic, or instance variable.
- A function call using any of the following functions:

car
cdr
caar
cdar
cadr
cddr
caaar
cdaar
cadar
cddar
caadr
cdadr
caddr
cdddr
caaaar
cdaaar
cadaar
cddaar
caadar
cdadar
caddar
cdddar
caaadr
cdaadr
cadadr
cddadr
caaddr
cdaddr
cadddr
cadddr
nthcdr
get
getf
gethash
aref
svref
first
second
third
fourth
fifth
sixth
seventh
eighth
ninth
tenth
nth
elt
symbol-plist
symbol-value
symbol-function
array-leader
fill-pointer
fdefinition
sys:function-spec-get
documentation
rest
rest1
rest2
rest3
rest4

- A function call to an access operation produced by **defstruct** (see paragraph 10.2, The **defstruct** Macro).

- A function call using any of the following functions, provided that the value of *form* is of the corresponding type for that function.

Function	Required Type
char	**string-char**
schar	**string-char**
bit	**bit**
sbit	**bit**
subseq	**sequence**

- A function call using any of the following functions, provided that the corresponding argument could serve as a *place* argument to **setf**. Also listed is the particular function that is applied to perform the assignment for **setf**.

Function	Argument Modified	Function Applied
char-bit	*char*	**set-char-bit**
ldb	*integer*	**dpb**
mask-field	*integer*	**deposit-field**

Consider the following example:

```
(setf x #\a) => #\a
(setf (char-bit x :HYPER) t) => t
(char-bit x :HYPER) => true
```

- A **the** type declaration form in which the *form* argument for **setf** is set to the declaration. For example:

```
(setf (the integer (cadr x)) (+ y 3))
```

This form is treated the same as the following:

```
(setf (cadr x) (the integer (+ y 3)))
```

- An **apply** invocation in which the first argument is of the form (**function** *access-function*) or is of the form **#'***access-function*, where *access-function* is recognized by **setf** as a *place* form. Note that the last argument appearing in *access-function* must also be the last argument in the update form. The reason for this requirement is that **apply** can handle a varying number of arguments, and therefore the update function must also handle a varying number of arguments. The only way to deal with this situation is to have the access and update functions expect their last forms to be the same regardless of the actual number of arguments. For example:

```
(setf (apply #'access-function arg1 arg2 ...arg-n last-arg)
                                                   new-value)
```

The preceding form must expand to the following:

```
(update-form item1 item2 ... item-m last-arg)
```

In this example, `last-arg` is the same in both forms, and `new-value` in the first form is the same as one of the items in the second form.

In practice, only **aref** satisfies the previously described requirement if **aset** is chosen as the update function. For example:

```
(setf (apply #'aref some-vector '(0)) 100)
```

The preceding form expands into the following:

```
(apply #'aset 100 some-vector '(0))
```

This form stores the value 100 in element 0 of `some-vector`. You can define other access and update functions that work with **apply** by using **defsetf**; however, these forms still must conform to the requirements placed on the last argument.

- A macro call such as the **setf** macro which expands the macro and analyzes the expansion code. If the macro call is not an acceptable *place* form, the error condition **sys:unknown-setf-reference** is signaled.
- A call to a function that was defined by the **defsubst** form and that expands to an acceptable *place* form.
- A **values** form. Each of the variable names that appear as arguments to the **values** function is assigned a new value. In this case, **setf** operates much the same way as **multiple-value-setq**, except that the variables can be generalized variables. The second argument to the **setf** should be a form that produces multiple values. If more values than variables are specified, then the extra variables are ignored. If not enough values are supplied, then the remaining variables are set to **nil**. For example:

```
(setf (values (aref quotient-array an-index)
              (aref remainder-array an-index))
      (floor 11 4))
```

In this example, the `floor` function produces two values: `2` and `3`. Therefore, the effect of the `setf` is the same as the following:

```
(setf (aref quotient-array an-index) 2)
(setf (aref remainder-array an-index) 3)
```

- Any form that looks like an access to a flavor instance variable. In this case, **setf** generates the appropriate update syntax. For example:

```
(setf (send foo :bar) new-value)
```

The preceding form is then expanded to the following:

```
(send foo :set-bar new-value)
```

Note that it is not possible for the compiler to know that the *place* argument is really a flavor instance access function. For the purposes of **setf**, it is sufficient that it merely looks like one.

- Any form for which there has been a **defsetf** or **define-setf-method** declaration.

You can define new ways for **setf** to expand by using **defsetf**.

psetf *{place value}** *[c]* Macro

This macro is like **setf**, except that the assignments of *values* to *places* are performed in parallel. All subforms are evaluated from left to right; after all the subforms have been evaluated, values are assigned in an unpredictable order. Thus, **psetf** may produce unexpected results if two or more *place* arguments refer to the same memory location.

The returned value of **psetf** is always **nil**.

shiftf *{place}+ value* *[c]* Macro

This macro sets the first *place* to the value from the second *place*, the second *place* to the value from the third *place*, and so on (a shift left). The *place* arguments can be anything allowable as a generalized variable for **setf**. The argument *value* does not have to be a generalized variable acceptable to setf, and its value is shifted to the last *place*. The original value of the first *place* is returned. For example:

```
(setq x (list 'a 'b 'c 'd 'e))=> (a b c d e)
(shiftf (second x) (third x) (fourth x) 'z) => b
x => (a c d z e)

(shiftf (second x) (cddr x) 'q) => c
x => (a (d z e) . q)
```

rotatef *{place}** *[c]* Macro

This macro is like **shiftf**, but the value of the last *place* is set to the value from the first *place* (a shift left circular). In other words, **rotatef** sets the first *place* to the value from the second *place*, the second *place* to the value from the third *place*, and so on until the last *place*. The last *place* then is set to the value from the first *place*. The returned value is **nil**. For example:

```
(setq x (list 'a 'b 'c 'd 'e))=> (a b c d e)

(rotatef (second x) (third x) (fourth x))=> nil
 x => (a c d b e)

(rotatef (first x) (second x)) => nil
x => (c a d b e)
```

defsetf *access-fn update-fn [doc-string]* *[c]* Macro
defsetf *access-fn lambda-list (store-variable)* *[c]* Macro
*{declaration | doc-string}** *{body-form}**

This macro defines the translation for the **setf** operation on a generalized variable specified by the argument (*access-fn arg*). The argument *access-fn* must be a function or macro name. The update function supplied (or defined via *body-forms*) performs the logical update and must also return the new value to be consistent with the **setf** definition.

The simplest situation in which to use **defsetf** is when there is an update function that does all the work of storing a value into the appropriate place and has the proper calling conventions:

(**defsetf** *function update-fn*)

The preceding form provides a translation that tells **setf** how to store into the following generalized variable:

(*function args...*)

This storage is performed by invoking a form such as the following:

(*update-fn args... new-value*)

Note that *new-value* must be the last item in the list of arguments to *update-fn*.

The more general form of **defsetf** is used when there is no setting function with exactly the right calling sequence. Thus, the *body-forms* tell **setf** how to store a value into the generalized variable (*function args...*) by providing something like a macro definition that expands into code and performs the actual storing. The *body-forms* compute the code, and the last of the *body-forms* returns a suitable expression.

The argument *lambda-list* should be a lambda list, which can have &optional and &rest parameters. When you use the backquote facility (described in Section 18, Macros), the *body-forms* should substitute (using the comma syntax) the values of the parameters in this lambda list in order to refer to the arguments in the **setf** calling form. Likewise, the *body-forms* should substitute the variable *store-variable* in order to refer to the value being stored.

Consider the following example:

```
(defun access-nth (index list)
   (nth index list))

(defun update-nth (list index new-value)
   (rplaca (nthcdr index list) new-value))

(defsetf access-nth (index list) (new-value)
   `(progn (update-nth ,list ,index ,new-value)
           ,new-value))

(setq sample-list '(a b c d))

(access-nth 2 sample-list) => c

(setf (access-nth 2 sample-list) 'z) => z

sample-list => (a b z d)

(macroexpand '(setf (access-nth 2 sample-list) 'z))
=> (progn (update-nth sample-list 2 'z) 'z)
```

In fact, the values bound to the lambda-list parameters and the store variable are not the actual subforms of the **setf** calling form; instead, they are **gensyms**. After the *body-forms* return, the corresponding expressions may be substituted for the gensyms, or the gensyms may remain as local variables with a suitable **let** provided to bind them. This procedure is how **setf** ensures a correct order of evaluation.

define-modify-macro *name lambda-list function* [*doc-string*] *[c]* Macro

This macro defines a macro that modifies its argument (such as **incf**).

name — The *name* argument is the name given to the macro being defined.

lambda-list — The *lambda-list* argument is a list of all arguments—except the first—accepted by the new macro. The first argument to the new macro is the equivalent of the *place* argument to **setf**. The *lambda-list* argument accepts only the lambda-list keywords &optional and &rest (which prevents the need for &key).

function — The *function* argument (sometimes called the *combiner function*) is applied to the original place value, along with any values specified in *lambda-list*, to produce the new value, which is stored back in the place of the original value (as with **setf**).

doc-string — The *doc-string* argument is the documentation string describing the new macro.

For example, the macro **incf** can be defined as follows:

```
(define-modify-macro incf (&optional (delta 1)) +)
```

If the `incf` macro had been defined in this way, the expansion would be as follows:

```
(macroexpand '(incf x 3)) => (setq x (+ x 3))
```

define-setf-method *access-fn lambda-list* {*declaration* | *docstring*}* {*body-form*}* *[c]* Macro

This macro defines the **setf** operation on a generalized variable accessed by the function specified by the *access-fn* argument. This function provides more power and generality than **defsetf** provides but is also more complicated to use.

NOTE: The use of the term *method* in this description is generic and has no relation to Explorer flavor methods.

The **define-setf-method** form receives its arguments almost like an analogous **defsetf**. However, the values it receives are the actual subforms and the actual form for the value, rather than gensyms that stand for them. The parameters of *lambda-list* are bound to the actual subforms of the *place* argument in the **setf** calling form, and the full power of **defmacro** lambda lists can be used to match against it.

The *body-forms* are once again evaluated, but **define-setf-method** does not return an expression to do the storing. Instead, it returns five values that contain sufficient information to enable anyone to examine and modify the contents of the *place*. This information tells the caller which subforms of the *place* need to be evaluated and how to use them to examine or set the value of the *place*. (Generally, the *lambda-list* is arranged to make each parameter receive one subform.) A temporary variable must be found or made (usually with **gensym**) for each subform. Another temporary variable should be made to correspond to the value to be stored. The following five returned values are everything that the macros used for manipulating generalized variables (**setf** or something more complicated) need to know to decide what to do:

- A list of the temporary variables for the subforms of the *place*, usually gensyms.
- A list of the subforms to which the temporary variables correspond. Usually, these variables are the evaluated lambda-list variables.
- A list of the temporary variables for the values to be stored, usually gensyms. Currently, there can only be one value stored; therefore, there is always only one variable in this list.
- A form to do the storing. This form refers to some or all of the temporary variables mentioned previously.
- A form to retrieve the value of the *place*. The **setf** form does not need to perform this retrieval, but **push** and **incf** do. This form should refer only to the temporary variables. To avoid causing this form to be evaluated, it should not contain any piece of the *place* being stored in.

For an example use of **define-setf-method**, see the example for the related function **get-setf-method**.

delete-setf-method *access-fn* *[c]* Function

This function removes the **setf** definition for *access-fn*. The *access-fn* argument must currently be defined as a setfable access form.

get-setf-method *form* *[c]* Function

This function invokes the **setf** method for *form* (which must be a list) and returns the five values produced by the body of the **define-setf-method** for the symbol that is the first element of *form*. The meanings of these five values are given in the previous description of **define-setf-method**. If the **setf** definition of an access operation was defined with **defsetf**, you still get five values, which can be interpreted as outlined previously in the description of **define-setf-method**. Thus, **defsetf** is an abbreviation for a suitable **define-setf-method**.

There are two ways to use **get-setf-method**. One way is in a macro that, like **setf**, **incf**, or **push**, wants to store into a *place*. The other way is in a **define-setf-method** for something like **ldb**, which performs a **setf** operation by setting one of its arguments. You append your new temporary variables and temporary arguments to those returned from **get-setf-method** to produce the combined lists that you return. Thus, the forms returned by the **get-setf-method** are placed into the forms you return.

An example of a macro that uses **get-setf-method** is **pushnew**. (The real **pushnew** is more complicated than that shown in the following example because it must handle the **:test**, **:test-not**, and **:key** arguments.) For example:

```
(defmacro pushnew (value place)
   (multiple-value-bind
         (tempvars tempargs storevars storeform refform)
         (get-setf-method place)
     (sys:sublis-eval-once
       (cons `(-val- . ,value) (pairlis tempvars tempargs))
       `(if (member -val- ,refform :test #'eq)
            ,refform
            ,(sublis (list (cons (car storevars)
                                 `(cons -val- ,refform)))
                      storeform))
       t t)))
```

In this example, if the value is already a member of the list (that is, the `,refform`), then the list is simply returned. If the value is not a member, then the `sublis` form changes every item in `storeform` that currently contains the list to be the cons of the new item onto the list.

An example of a **define-setf-method** that uses **get-setf-method** is that for **ldb**:

```
(define-setf-method ldb (bytespec int)
  (multiple-value-bind
      (temps vals stores store-form access-form)
      (get-setf-method int)
      (let ((btemp (gensym))
            (store (gensym))
            (itemp (first stores)))
            (values (cons btemp temps)
                    (cons bytespec vals)
                    (list store)
                   `(progn
                       ,(sublis
                          (list (cons itemp
                                      `(dpb ,store ,btemp
                                            ,access-form)))
                          store-form)
                      ,store)
                   `(ldb ,btemp ,access-form)))))
```

This example primarily demonstrates that **setf** methods must be written in such a way that they can interact successfully with other **setf** forms. You never know how indirect or how abstract the *place* argument may be, but as long as all programmers follow the style of the preceding example, the setf methods should work properly. Specifically, the variable names and the value lists must be kept in sync, and the update (and access) forms must use these variable names.

get-setf-method-multiple-value *form* *[c]* Function

This function is similar to **get-setf-method** but does not concern itself with how many variables it stores. The **get-setf-method-multiple-value** function returns the five values of the **define-setf-method** with *form* as an argument. This argument must be a generalized variable meeting the requirements for the *place* argument to **setf**. The **get-setf-method-multiple-value** function should be used for storing multiple values in a generalized variable. Currently, Common Lisp has no situations requiring such a function, but **get-setf-method-multiple-value** has been defined to provide for future extensions.

Logical Values and Symbol Predicates

2.14 The following constants represent logical values. The following functions are predicates used to test symbols.

t *[c]* Constant

The value of this constant should not be changed; it represents the logical value *true*.

nil *[c]* Constant

The value of this constant should not be changed; it represents the logical value *false*. The **nil** constant also stands for the empty list and as such can be written `()`.

symbolp *object* *[c]* Function

This predicate returns true if *object* is a symbol; otherwise, it returns **nil.** Note the following equivalence:

```
(symbolp sign) <=> (typep sign 'symbol)
```

nsymbolp *object* Function

This function returns **nil** if *object* is a symbol; otherwise, it returns true.

keywordp *object* *[c]* Function

This predicate returns true if the argument is a symbol that is a keyword (in other words, the symbol belongs to the KEYWORD package). Keywords are treated like constants in that, when evaluated, they return themselves (see **constantp,** paragraph 13.5, Global Variables and Named Constants). The argument *object* can be any Lisp object.

boundp *symbol* *[c]* Function
variable-boundp *symbol* Special Form

The **boundp** predicate returns true if the value cell of *symbol* is not empty; otherwise, it returns **nil.** For example:

```
(defvar x)                          ; Proclaim x special.
(setf x 1)
(and (boundp 'x) x) => 1
(and (boundp 'y) y) => nil
```

The **variable-boundp** special form is similar to **boundp,** but its argument is not evaluated and can be a lexical variable or an instance variable as well as a special variable.

The form `(and (boundp 'x) x)` returns the value of `x` if it is defined. Since it is defined, the form returns the non-**nil** value `1`.

boundp-globally *symbol* Function

This function returns true if the global binding of *symbol* is bound.

3

NUMBERS

Number Definitions

3.1 The three general types of numbers are *rational*, *floating-point*, and *complex* numbers. While each type possesses special qualities that make it useful for certain kinds of processing, most numerical functions operate on any of these types without any special notation.

Common Lisp allows some latitude in the implementation of numbers. Portable programs cannot assume that numbers are conventional data objects. Thus, the **eq** function may not reliably operate on numbers. For example:

```
(let ((yabba num)
      (dabba num))
   (eq yabba dabba))
```

If `num` is a number, this expression may not necessarily return true. However, the Explorer system implements numbers such that the preceding example would return true if `num` is of type **fixnum** or **short-float**. Thus, for portable Common Lisp programs, use the numeric comparison functions (described later in this section) when all the arguments are known to be numbers, and use **eql** to test for identity when one of the arguments may not be a number.

Rational Numbers

3.1.1 The number type **rational** is made up of two types of numbers: *integers* and *ratios*. Numbers of type **integer** are intended to behave as mathematic integers. Theoretically, there is no limit on the size of an integer number on the Explorer. The system automatically makes allowances to represent rational numbers of any size. However, the actual maximum integer size is limited by your system's virtual address space to represent this number, and, of course, manipulating such large numbers reduces the quality of your system's performance. Integers are written as follows:

[sign]{digits}$^{+}$

Integers can be represented internally as either fixnums or bignums. **Fixnums** are a type of integer that is more efficient than arbitrarily large integers, but their magnitude is limited. The constants **most-negative-fixnum** and **most-positive-fixnum** define the range of fixnums. On the Explorer system, these constants are equal to -2^{24} and $2^{24}-1$, respectively. These limits are derived from the size of a word on the Explorer and may vary with other Common Lisp implementations. All integers that cannot be represented as fixnums are **bignums**. Unless you explicitly test for this distinction, the difference between fixnums and bignums will be transparent to you and is apparent only when efficiency of representation is important.

A **ratio** is a type of number whose value is the mathematical quotient of two integers. A ratio consists of a signed integer called the numerator and a positive integer called the denominator. Rational numbers (that is, ratios) are written as follows:

[sign]{numerator digits}$^{+}$/{denominator digits}$^{+}$

In this format, the denominator digits cannot all be zero and no spaces are permitted around the slash (/).

In a rational number's canonical representation, all of the common factors are removed from the numerator and denominator. If the denominator is 1, the number is converted to an integer.

Computations involving rational numbers always produce rational numbers in their canonical form. Also, rational numbers are always printed in canonical form.

Controlling Radices

3.1.2 When the Explorer system reads or writes a number, it uses the default radices established by the variables ***read-base*** and ***print-base***, respectively. These variables are initially set to 10. but can be changed or temporarily bound to another value. A period in a number is interpreted as a decimal point so that the number is read in base 10 regardless of the value of ***read-base***.

NOTE: A trailing period denotes a decimal integer, *not* a floating-point number as in some other languages.

You can override the default radix for reading by using the following notation:

*#dd***R***nnnnn*

The digits between the # and the **R** specify the radix in which to read the number *nnnnn*. The radix *dd* must be an unsigned decimal integer in the range of 2 to 36 inclusive. The characters used to represent the digits *nnnnn* must be limited to those appropriate for that base. For instance, numbers in base 8 can use only the characters 0 through 7. Bases that require supplemental characters beyond the decimal set use letters of the alphabet, beginning with *a*. These alphabetic characters can be in either uppercase or lowercase. Note that in the case of ratios, the radix applies to the numerator and denominator and the slash is allowed as part of the number specification. For example:

```
#8R10/16 => 4/7

;The preceding is the same as the following:
8/14 => 4/7
```

The use of binary, octal, and hexadecimal numbers is so common that the following special abbreviations are made available:

Radix	Standard Representation	Abbreviation
binary	#2R*nnnnn*	#B*nnnnn*
octal	#8R*nnnnn*	#O*nnnnn*
hexadecimal	#16R*nnnnn*	#X*nnnnn*

Note that since the Reader maps lowercase characters to uppercase, you can also use **r**, **b**, **o**, and **x** to specify radices.

Floating-Point Numbers

3.1.3 Floating-point numbers represent mathematical real numbers. Common Lisp defines the type **float** as being made up of four kinds of floating-point numbers: **short-float**, **long-float**, **single-float**, and **double-float**. Of these types, **short-float** floating-point numbers are for optimum speed and storage space, **long-float** floating-point numbers are for optimum precision, and **single-float** and **double-float** floating-point numbers provide precisions somewhere between the **short-float** and **long-float** formats.

On the Explorer system, floating-point numbers are represented in accordance with IEEE standard 754 (although **short-float** is not officially part of the standard, it is treated in a fashion that is a logical extension of IEEE 754). Note that the feature **ieee-floating-point-format** is present in the ***features*** variable (which is described in the *Explorer Input/Output Reference* manual).

These four types of floating-point numbers are not necessarily distinct, though all must be included in any Common Lisp system. For instance, the Explorer has three distinct implementations for floating-point numbers in which the types **double-float** and **long-float** map to the same implementation scheme. Other Common Lisp implementations may have more or fewer implementation schemes and may map the float types to implementations in a way different from the Explorer.

The notation for floating-point numbers is either decimal fraction or computerized scientific notation. The syntax is defined as follows:

[sign] $\{\text{digit}\}^{*}.\{\text{digit}\}^{+}$ [float-type [sign] $\{\text{digit}\}^{+}$]

or:

[sign] $\{\text{digit}\}^{+}$ [.$\{\text{digit}\}^{*}$] float-type [sign] $\{\text{digit}\}^{+}$

where:

sign is + or –
digit is 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9
float-type is E, S, F, D, or L

The letters **S**, **F**, **D**, **L** explicitly identify the floating-point types **short-float**, **single-float**, **double-float**, and **long-float**, respectively. These characters can be either uppercase or lowercase. When the float-type is **e**, **E**, or unspecified, then the variable ***read-default-float-format*** specifies the type of floating-point number to be used. The initial value of this variable is **single-float**. Floating-point numbers are always read in base ten. Although you can syntactically specify another radix, this radix is overridden once the number is found to be of type **float**.

If you supply a number that ends in a decimal point, such as 10., it is treated as an integer and not as a floating-point number. If you want to express a floating-point number, you must supply at least one fractional digit after the decimal point, such as 10.0, or a **float** type specifier.

Regardless of how a number is expressed, the internal representation is usually a normalized version of that number. IEEE standard 754 allows for denormalized numbers to implement gradual underflow (which the Explorer system does not currently support). Specifically, the number (in binary) is adjusted to isolate the most-significant *n* bits, where *n* is the size of the mantissa. For each bit, the number is adjusted right or left and the exponent is incremented or decremented accordingly. If the exponent does not fit within the stated field size for the intended floating-point type, an error is signaled. If significant bits of the mantissa do not fit within the stated field size, then the least-significant bits are used to round the number up or down. On the Explorer system, the four floating-point types have the field characteristics specified in Table 3-1.

Table 3-1

Field Characteristics for Floating-Point Types

Data Type	Exponent Size*	Range of Exponent	Mantissa Size*	Decimal Digits of Precision
short-float	8	$10^{\pm 38}$	17	5
single-float	8	$10^{\pm 28}$	24	7
double-float	11	$10^{\pm 308}$	53	16
long-float	11	$10^{\pm 308}$	53	16

Note:

* The exponent and mantissa sizes are expressed in bits.

Currently, **long-float** is implemented as **double-float** on the Explorer.

Complex Numbers

3.1.4 The data type **real** consists of all numbers that do not have a complex component, such as fixnums, bignums, floating-point numbers, and rational numbers. Complex numbers, numbers of type **complex**, are composed of two parts (a real part and an imaginary part) and are written in the following format:

#C(*real-part imaginary-part*)

The real part and the imaginary part can be of any numeric type except **complex**, but both parts must be of the same type. Specifically, both parts must be rational or both parts must be of the same floating-point type. If the two parts are not of the same type, then one of the numbers is converted according to the rules of floating-point contagion. (Contagions are discussed in the next numbered paragraph.)

The type specifier for a complex number is represented as a list in which the first element is the type name **complex** and the second element is the type of the component parts. For example:

```
(typep #c(1 2) ′(complex rational)) => true
(typep #c(1.5 2.5) ′(complex single-float)) => true
```

A canonical complex number whose components are rational can never have a value of 0 for its imaginary part. If some computation should derive such a number, the number is immediately converted to a rational number equal to the real part. However, complex numbers with floating-point components can have an imaginary part of 0.0.

Precision, Coercion, Contagion, and Canonicalization

3.1.5 *Precision* can be regarded as potential accuracy. That is, for floating-point numbers, the more bits allocated for retaining the fractional part, the higher the probability of obtaining an accurate representation. However, inaccuracy is necessarily introduced because most mathematical real numbers will overflow any practical boundaries that you establish. Given the preceding definition for rational numbers, however, the virtual address space is the only boundary that will overflow.

Numerical coercion is the process of converting a number of one type to a number of another type. You can explicitly perform this conversion using various support functions. Typically, you coerce a number to another type to simplify or speed up a calculation. For example, you could coerce a floating-point number to an integer inside a looping construct to speed up the construct's execution. Some precision might be lost in making this coercion, but in some cases speed may be more important than precision.

Coercion is also implicitly invoked within the context of your program and even in the run-time type of your symbols. This implicit conversion is called *contagion*. Whenever a numerical operation requires that its arguments be of the same type, the more specific type is converted to the more general type. For instance, a rational number is converted to a floating-point number, a short-precision float is converted to a longer-precision float, and floating-point numbers are converted to complex numbers.

Generally speaking, the result of any numeric operation of similar types (rational, floating-point, or complex) will return a value of a similar type that allows for the highest degree of accuracy. Note that although the multiplication of two fixnums may produce a bignum, this product is still of type **integer** and the precision simply increases as necessary to retain accuracy. For floating-point numbers, however, the precision type of the result is not increased because it is assumed that this would not increase accuracy. For example, multiplying two single-floats produces another single-float, not a double-float or a long-float.

Some numeric values are automatically coerced into a less general yet equally accurate type. The general term for this is *canonicalization*. As mentioned previously, a canonical ratio with a denominator of 1 is converted to an integer, and a rational complex number with an imaginary part of 0 is converted to a rational number. These particular conversions are made for efficiency, although all types of numbers are not automatically converted to their simplest possible type. For instance, numbers such as 14.0 are not converted into 14. because floating-point numbers are not assumed to be completely accurate and can therefore never be converted back to rationals unless explicitly requested by the user.

Number Constants

3.2 The following constants can be used to provide numeric boundary values.

most-positive-fixnum *[c]* Constant
most-negative-fixnum *[c]* Constant
most-positive-short-float *[c]* Constant
most-negative-short-float *[c]* Constant
least-positive-short-float *[c]* Constant
least-negative-short-float *[c]* Constant
most-positive-single-float *[c]* Constant
least-positive-single-float *[c]* Constant
most-negative-single-float *[c]* Constant
least-negative-single-float *[c]* Constant
most-positive-double-float *[c]* Constant
least-positive-double-float *[c]* Constant
most-negative-double-float *[c]* Constant
least-negative-double-float *[c]* Constant
most-positive-long-float *[c]* Constant
least-positive-long-float *[c]* Constant
most-negative-long-float *[c]* Constant
least-negative-long-float *[c]* Constant

These constants specify the positive and negative numbers of the greatest and least magnitude that can be represented by the type indicated.

short-float-epsilon *[c]* Constant
single-float-epsilon *[c]* Constant
double-float-epsilon *[c]* Constant
long-float-epsilon *[c]* Constant

These constants specify, for the various floating-point number formats, the smallest positive number that can be added to a floating-point 1 to produce a noticeable change from the value of 1. For example, if *x* equals 1.0s0, a number smaller than **short-float-epsilon** added to *x* returns 1.0s0.

short-float-negative-epsilon *[c]* Constant
single-float-negative-epsilon *[c]* Constant
double-float-negative-epsilon *[c]* Constant
long-float-negative-epsilon *[c]* Constant

These constants specify, for the various floating-point number formats, the smallest positive number that can be subtracted from a floating-point 1 to produce a noticeable change from the value of 1.

Number Comparisons

3.3 The following functions take one or more arguments, which all must be numbers. For all of these functions (except **max** and **min**), if the sequence of arguments satisfies the function's specified test, then true is returned; otherwise, **nil** is returned.

Only = and /= accept complex numbers (the other functions take only non-complex arguments).

= *number* &rest *more-numbers* *[c]* Function

This is the numeric *equal* test; all arguments must be equal in value. If the arguments are complex numbers, this function returns true for any two complex numbers whose real parts are equal (according to =) and whose imaginary parts are equal.

/= *number* &rest *more-numbers* *[c]* Function

This is the numeric *not equal* test; no two arguments are equal in value. If the arguments are complex numbers, this function returns true for any two complex numbers whose real parts are not equal (according to /=) or whose imaginary parts are not equal.

< *number* &rest *more-numbers* *[c]* Function

This function is the numeric *less than* test; all arguments (from left to right) must be monotonically increasing in value.

> *number* &rest *more-numbers* *[c]* Function

This function is the numeric *greater than* test; all arguments must be monotonically decreasing in value.

<= *number* &rest *more-numbers* *[c]* Function

This function is the numeric *less than or equal* test; all arguments must be monotonically nondecreasing in value.

>= *number* &rest *more-numbers* *[c]* Function

This function is the *greater than or equal* test; all arguments must be monotonically nonincreasing in value.

max *number* &rest *more-numbers* *[c]* Function
min *number* &rest *more-numbers* *[c]* Function

These functions require noncomplex numbers for arguments. The **max** function returns the argument with the largest value. The **min** function returns the argument with the smallest value. For example:

```
(max 59 58 57) => 59               (min 59 58 57) => 57
(max -22 -33 -3) => -3             (min -22 -33 -3) => -33
(max 0 -11) => 0                   (min 0 -11) => -11
(max 4) => 4                       (min 4) => 4
(max 14.302 14.301 15) => 15       (min 14.302 14.301 15) => 14.301
```

Arithmetic

3.4 The following functions perform the standard arithmetic operations. All arguments must be numbers.

+ &rest *numbers* *[c]* Function

This function returns the sum of *numbers*. If no arguments are supplied, it returns 0, the identity for addition.

- *number* &rest *more-numbers* *[c]* Function

This function can be used in two ways—either to negate *number* if *more-numbers* are unspecified or to subtract each element of *more-numbers* from *number*. For example:

```
(- 5) => -5
(- 10 3 2 1) => 4
;The preceding form is equivalent to the following.
(- (- (- 10 3) 2) 1) => 4
```

***** &rest *numbers* *[c]* Function

This function successively multiplies each number in *numbers* by the product of the numbers preceding it. Thus, `(* 1 2 5)` returns 10. If *numbers* is unspecified, this function returns 1, the identity for multiplication.

/ *number* &rest *more-numbers* *[c]* Function

This is the division and reciprocal operation. The Common Lisp function / can be used in two ways. If *more-numbers* are unspecified, / returns the reciprocal of *number*. Otherwise, *number* is divided by the first number in *more-numbers*; then this quotient is divided by the next number in *more-numbers*, and so on until all numbers in *more-numbers* are used.

A ratio is produced if the mathematical quotient of two integers is not an exact integer. Consider the following examples:

```
(/ 6 2) => 3
(/ 5 2) => 5/2
(/ 5.0 2) => 2.5
(/ 3) => 1/3
(/ 24 2 3) => 4
```

In Common Lisp, use one of the functions **floor**, **ceiling**, **truncate**, or **round** to divide one integer by another to produce an integer result. If *number* or any element of *more-numbers* is a floating-point number, the rules of floating-point contagion are used in producing the returned value. Note that this is different in Zetalisp mode.

quotient *number* &rest *numbers* Function

This function performs division but differs from the Common Lisp function / in the following ways:

- If only one argument is supplied, it is returned unchanged.
- If both arguments are integers, an integer result is returned, with any fractional part discarded.

Consider the following example:

```
(quotient 5 2) => 2
```

This function is supported for the sake of old programs; new programs should use / or **truncate**, as appropriate.

1+ *number* *[c]* Function

This is the incrementor function. It returns a number equal to *number*+1. For example:

```
(1+ x) <=> (+ x 1)
```

1– *number* *[c]* Function

This function returns its argument decremented by 1. Note that this function signifies *number–1*, not *1–number*. Note the following equivalence:

```
(1- x) <=> (- x 1)
```

incf *place* &optional *amount* *[c]* Macro
decf *place* &optional *amount* *[c]* Macro

The **incf** macro increments the value at *place* by *amount*, which defaults to 1. The **incf** form returns the new value of *place* after the addition. The **decf** macro performs the same operation, but instead of incrementing, it decrements. For example:

```
(setf thing 13) => 13
(incf thing) => 14
(decf thing 23) => -9
thing => -9
(decf thing -5) => -4
(decf thing) => -5
thing => -5
```

The form (**incf** *place amount*) is like (**setf** *place* (+ *place amount*)), but **incf** evaluates *place* only once. Furthermore, **incf** may be more efficient than **setf** for some *place* subforms.

conjugate *number* *[c]* Function

This function returns the complex conjugate of its argument. If the argument is noncomplex, then this function simply returns the argument. For example:

```
(conjugate #C(1/2 -2/3)) => #C(1/2 2/3)
(conjugate #C(2.0S3 1.0S3)) => #C(2000.0s0 -1000.0s0)
(conjugate 1.5) => 1.5
```

mod *number divisor* *[c]* Function

This function returns the root of *number* modulo *divisor*. This is a number between 0 and *divisor*, or possibly 0, whose difference from *number* is a multiple of *divisor*. It is the same as the second returned value of the form (**floor** *number divisor*). For example:

```
(mod 7 15) => 7
(mod -7 15) => 8
(mod -7 -15) => -7
(mod 7 -15) => -8
(mod 15 7) => 1
```

rem *number divisor* *[c]* Function

This function returns the remainder of *number* divided by *divisor*, which is the same as the second returned value of the form (**truncate** *number divisor*). Both *number* and *divisor* can be integers or floating-point numbers. Consider the following example:

```
(rem 7 15) => 7
(rem -7 15) => -7
(rem -7 -15) => -7
(rem 7 -15) => 7
(rem 15 7) => 1
```

gcd &rest *integers* *[c]* Function

This function returns the greatest common divisor of its arguments, which must be integers. With no arguments, **gcd** returns 0. If one argument is passed, the absolute value of the integer is returned. For example:

```
(gcd 36 60) => 12
(gcd 45 54 -81) => 9
(gcd 6) => 6
(gcd -3) => 3
(gcd) => 0
```

lcm *integer* &rest *integers* *[c]* Function

This function returns the least common multiple of the specified integers. At least one argument must be provided, and if one or more arguments are equal to 0, then the function returns 0. If only one argument is provided, then the function returns its absolute value. The operation of this function can be described as follows:

```
(lcm x y) <=> (/ (abs (* x y)) (gcd x y))
```

Consider the following example:

```
(lcm 1 2 3) => 6
(lcm 1 2 3 4) => 12
(lcm 1 2 3 4 5) => 60
```

abs *number* *[c]* Function

This function returns the absolute value of *number*. If *number* is complex, a real value equivalent to the following form (though not necessarily computed in this way) is returned:

```
(sqrt (+ (expt (realpart number) 2) (expt (imagpart number) 2)))
```

Exponential and Logarithmic Functions

3.5 The following functions perform exponential and logarithmic operations. Those functions dealing with the base of the natural logarithms convert all arguments to floating-point numbers and return a single-precision floating-point number.

exp *power* *[c]* Function

This function raises *e* to the power of *power*, where *e* is the base of the natural logarithms.

expt *number power* *[c]* Function

This function returns *number* raised to the power of *power*. The result is rational (and possibly an integer) if *number* is rational and *power* is an integer. If *power* is an integer, a repeated-multiplication algorithm is used. If *power* is 0, then the result is the number 1 in the type of whatever type *number* is. It is an error, however, for *number* to be 0 when *power* is non-integer 0. Consider the following equivalence:

```
(expt x y) <=> (exp (* y (log x)))
```

log *number* &optional *base* *[c]* Function

This function returns the logarithm of *number* in the base of *base*. If a *base* argument is not provided, then the function defaults the base to *e* (see exp), the base of the natural logarithms. For example:

```
(log 27.0 3) => 3.0
(log 100.0 10) => 2.0
```

This function can return a complex value if *number* is not complex but is negative.

sqrt *number* *[c]* Function

This function returns the square root of *number*. A mathematically unavoidable discontinuity occurs for negative real arguments, for which the value returned is a positive real number multiplied by *i*, which is represented as the imaginary part of a complex number. For example:

```
(sqrt 4) => 2.0
(sqrt -4) => #C(0.0 2.0)
(sqrt #C(-4 .0001)) => #C(2.5e-5 2.0)          ; approximately
(sqrt #C(-4 -.0001)) => #C(2.5e-5 -2.0)        ; approximately
```

isqrt *integer* *[c]* Function

This function is the integer square-root operation. The argument *integer* must be a nonnegative integer; the result is the greatest integer less than or equal to the exact square root of *integer*. For example:

```
(isqrt 16) => 4
(isqrt 17) => 4
(isqrt 228) => 15
```

Trigonometric and Related Functions

3.6 The following functions perform trigonometric and transcendental operations. Common Lisp requires that the arguments to the basic trigonometric functions (cos, sin, and tan) be specified in radians.

phase *number* *[c]* Function

This function returns the phase angle of the complex number *number* in its polar form. This is the angle in radians from the positive *x* axis to the ray from the origin through *number*. The value is always in the interval -pi to pi. For example:

```
(phase 4) => 0.0
(phase -4) => 3.1415927                        ; pi
(phase #C(-4 -.0001)) => 3.1415904             ; near -pi
(phase 0) => 0.0
```

signum *number* *[c]* Function

This function returns a number of the same type with unit magnitude and the same sign as *number*. If *number* is 0, the returned value is 0.

If *number* is rational, the returned value is 0, 1, or -1. If *number* is a floating-point number, the result is a floating-point number (0.0, 1.0, or -1.0) of the same type. If *number* is a complex number, the result has the same phase angle as *number* but is scaled to the unit circle. For example:

```
(signum 0) => 0
(signum 5/2) => 1
(signum #C(10 10))
=> #C(0.70710677 0.70710677)     ; 45 degree angle
(signum #C(0.0 -1986.0)) => #C(0.0 -1.0)
```

sin *radians* *[c]* Function
cos *radians* *[c]* Function
tan *radians* *[c]* Function
sind *degrees* Function
cosd *degrees* Function
tand *degrees* Function

The **sin**, **cos**, and **tan** functions return the sine, cosine, and tangent, respectively, of the value specified by *radians*.

The functions **sind**, **cosd**, and **tand** also return the sine, cosine, and tangent of the argument, but you must specify the argument in *degrees*.

cis *radians* *[c]* Function

This function returns the complex number of unit magnitude whose phase is *radians* (which must be a real number). This is equal to the following:

```
(complex (cos radians) (sin radians))
```

asin *numbers* *[c]* Function
acos *numbers* *[c]* Function

These functions return the angle in radians whose sine (or cosine) is equal to *numbers*. They can be defined as follows:

asin $-i \log (ix + \sqrt{1 - x^2})$

acos $-i \log (x + i\sqrt{1 - x^2})$

These functions can return a complex result if the absolute value of *numbers* is greater than 1.

atan *y* &optional *x* *[c]* Function

This function calculates the arc tangent of its arguments and returns the result in radians. The **atan** function can be defined as follows:

$$-i \log ((1 + iy)\sqrt{1/(1 + y^2)})$$

If only *y* is specified (which can be complex), the value is the angle, in radians, whose tangent is *y*. If the argument *y* is noncomplex, the result is also noncomplex and ranges between -pi/2 and pi/2.

If *x* is also given, the arguments must be real, and the result is an angle whose tangent is *y*/*x*. The signs of the two arguments are used to choose between two angles that differ by pi and have the same tangent. The returned value is the signed angle between the *x* axis and the line from the origin to the point (*x*, *y*) and is always between -pi (exclusive) and pi (inclusive). This is also the **phase** of (**complex** *x*/*y*). Table 3-2 shows various special cases of the result of **atan**.

Table 3-2 **Cases Involving atan**

x	y	Result
>0	0	0
>0	>0	0< result <pi/2
0	>0	pi/2
<0	>0	pi/2< result <pi
<0	0	pi
<0	<0	–pi< result <–pi/2
0	<0	–pi/2
>0	<0	–pi/2< result <0
0	0	Error

pi *[c]* Constant

This constant is equal to pi as a long floating-point number.

You can produce a value approximately equal to pi in another precision by using a floating-point number *num* of this precision in the form (**float pi** *num*). The same result can be achieved by specifying the type of precision in the form (**coerce pi** *float-type*).

sinh *number* *[c]* Function
cosh *number* *[c]* Function
tanh *number* *[c]* Function
asinh *number* *[c]* Function
acosh *number* *[c]* Function
atanh *number* *[c]* Function

These functions are the hyperbolic versions of **sin**, **cos**, **tan**, **asin**, **acos**, and **atan**. When these functions are provided with an argument x, they can be defined as follows:

sinh	Hyperbolic sine:	$(e^{x} - e^{-x})/2$
cosh	Hyperbolic cosine:	$(e^{x} + e^{-x})/2$
tanh	Hyperbolic tangent:	$(e^{x} - e^{-x})/(e^{x} + e^{-x})$
asinh	Hyperbolic arc sine:	$\log(x + \sqrt{1 + x^2})$
acosh	Hyperbolic arc cosine:	$\log(x + (x+1)\sqrt{(x - 1)/(x + 1)})$
atanh	Hyperbolic arc tangent:	$\log((1 + x)\sqrt{1 - 1/x^2})$

The functions acosh and **atanh** can return complex values even if *number* is a real value, if *number* is less than 1 for **acosh**, or if *number*'s absolute value is greater than 1 for **atanh**.

Standard Number Conversion

3.7 The following functions perform standard number conversions.

float *number* &optional *float* *[c]* Function

This function converts *number* to a floating-point number and returns it.

If *float* is specified, it must be a floating-point number, and the returned value is in the same floating-point format as *float*. If *number* is a floating-point number of a different format, then it is converted.

If *float* is omitted, the *number* is converted to a number of type **single-float** unless it is already a floating-point number.

A complex number is converted to another complex number whose real and imaginary parts are converted to the same floating-point format as *float* or, if *float* is omitted, to a number of type **single-float** unless they were already floating-point numbers. Note that this is an extension to the Common Lisp definition. See also **coerce**, paragraph 12.9, Type Conversion.

short-float *number* Function
double-float *number* Function

These functions convert *number* into short-float or double-precision floating-point numbers.

rational *number* *[c]* Function
rationalize *number* *[c]* Function
rationalize *number* &optional *precision* Function

The function **rational** returns *number* as a rational number. If *number* is an integer or a ratio, it is returned unchanged. If it is a floating-point number, it is regarded as an exact fraction whose numerator is the mantissa and whose denominator is a power of 2. For any other argument, an error is signaled. For instance, using the function **integer-decode-float**, you can see that 0.75 has a numerator of 1610612736 and a denominator of 2^{31}:

```
(/ 1610612736 (expt 2 31)) => 3/4
```

The function **rationalize** returns a rational approximation to *number*. If there is only one argument and it is an integer or a ratio, it is returned unchanged. If the argument is a floating-point number, a rational number is returned, which, if converted to a floating-point number, would produce the original argument. Of all such rational numbers, the one chosen has the smallest numerator and denominator.

If there are two arguments to **rationalize**, the second one specifies how many digits of the first argument should be considered significant. The argument *precision* can be a positive integer (the number of bits to use), a negative integer (the number of bits to drop at the end), or a floating-point number (which, minus its exponent, is the number of bits to use).

Also, when two arguments are provided to **rationalize** and the first is rational, the value is a *simpler* rational that is an approximation.

complex *real-part* &optional *imaginary-part* *[c]* Function

This function returns a complex number whose real part is *real-part* and whose imaginary part is *imaginary-part*. If *real-part* is rational and *imaginary-part* is 0 or omitted, the value is *real-part*. If *real-part* is a floating-point number and *imaginary-part* is 0 or omitted, a peculiar complex number is created whose numeric value is actually real. Note that **realp** of this peculiar complex number is **nil** even though the mathematical value is indeed real.

The value returned by **complex** can sometimes be a rational number rather than a complex number because of the rule of canonicalization of complex rationals.

ceiling *number* &optional *divisor* *[c]* Function
floor *number* &optional *divisor* *[c]* Function
truncate *number* &optional *divisor* *[c]* Function
round *number* &optional *divisor* *[c]* Function

With two arguments, the quotient of *number* divided by *divisor* is converted to an integer and returned. When these functions are provided with only one argument, the *divisor* argument defaults to 1. In this case, these functions convert the number of the argument to an integer, unless it already is one, in which case it is returned unchanged.

The function **ceiling** returns two values. The first value is the smallest integer greater than or equal to the quotient of *number* divided by *divisor*. The second returned value is the remainder, *number* minus *divisor* times the first returned value.

The function **floor** returns two values; the first is the largest integer less than or equal to the quotient of *number* divided by *divisor*. The second returned value is the remainder, that is, *number* minus *divisor* times the first returned value.

The function **truncate** is the same as **floor** if the arguments have the same sign. When the arguments have different signs, **truncate** operates the same as **ceiling**. The first returned value of **truncate** is the nearest integer, in the direction of 0, to the quotient of *number* divided by *divisor*. The second returned value of **truncate** is the remainder, that is, *number* minus *divisor* times the first returned value.

The function **round** returns two values: the first value is the nearest integer to the quotient of *number* divided by *divisor*. If the quotient is midway between two integers, the even integer of the two is used. The second returned value is the remainder, that is, *number* minus *divisor* times the first returned value. The sign of this remainder cannot be predicted from the signs of the arguments alone.

Table 3-3 shows the difference between these four functions when passed only one argument, that is, when the *divisor* argument defaults to 1.

Table 3-3

Values Returned by floor, ceiling, truncate, and round

	—— First Returned Value ——			
Argument	**floor**	**ceiling**	**truncate**	**round**
2.6	2	3	2	3
2.5	2	3	2	2
2.4	2	3	2	2
0.7	0	1	0	1
0.3	0	1	0	0
-0.3	-1	0	0	0
-0.7	-1	0	0	-1
-2.4	-3	-2	-2	-2
-2.5	-3	-2	-2	-2
-2.6	-3	-2	-2	-3

Nontrivial Floating-Point Conversion

3.8 The following functions convert numbers to floating-point format.

ffloor *number* &optional *divisor* *[c]* Function
fceiling *number* &optional *divisor* *[c]* Function
ftruncate *number* &optional *divisor* *[c]* Function
fround *number* &optional *divisor* *[c]* Function

These functions are like **floor, ceiling, truncate,** and **round** but return floating-point numbers.

If *number* is a floating-point number, then the result is the same type of floating-point number as *number*. For example:

```
(ffloor -7.875) => -8.0  0.125
```

Number Component Extraction

3.9 The following functions are used to extract components from nontrivial numbers.

realpart *number* *[c]* Function
imagpart *number* *[c]* Function

The function **realpart** returns the real part of the complex number *number*. If *number* is real, **realpart** simply returns *number*.

The function **imagpart** returns the imaginary part of the complex number *number*. If *number* is a rational, **imagpart** returns 0; if *number* is a floating-point number, **imagpart** returns a floating-point 0 of the same type as *number*.

numerator *number* *[c]* Function
denominator *number* *[c]* Function

The **numerator** function returns the numerator of the rational number *number*. If *number* is an integer, the returned value equals *number*. If *number* is not an integer or a ratio, an error is signaled.

The **denominator** function returns the denominator of the rational number *number*. If *number* is an integer, the value is 1. If *number* is not an integer or ratio, an error is signaled.

The **denominator** function always returns a positive integer; the **numerator** function returns both positive and negative integers. For example:

```
(numerator (/ 8 -3)) => -8
(denominator (/ 8 -3)) => 3
```

decode-float *float* *[c]* Function
integer-decode-float *float* *[c]* Function
scale-float *float integer* *[c]* Function
float-sign *float* &optional *identity* *[c]* Function
float-radix *float* *[c]* Function
float-digits *float* *[c]* Function
float-precision *float* *[c]* Function

These functions extract various numbers related to the argument *float*, which must be a floating-point number.

The function **decode-float** returns three values that describe the value of the argument *float*. The first returned value is a positive floating-point number of the same format having the same mantissa but with an exponent chosen to make it between ½ (inclusive) and 1 (exclusive). The second returned value is the exponent of *float*: the power of 2 by which the first value needs to be scaled in order to return *float*. The third returned value expresses the sign of *float*. It is a floating-point number that is of the same format as *float* and whose value is either 1 or -1. For example:

```
(decode-float 38.2) => 0.596875 6 1.0
```

The function **integer-decode-float** is like **decode-float**, except that the first returned value is scaled to make it an integer, and the second value (the exponent) is adjusted to compensate for the scaling. For example:

```
(integer-decode-float 38.2) => 10013901 -18 1.0
```

The function **scale-float** multiplies *float* by 2 raised to the *integer* power. For example:

```
(scale-float (float 10013901) -18) => 38.2
```

The function **float-sign** returns a floating-point number whose sign matches that of *float* and whose magnitude and format are those of *y* (which must be a floating-point number). If *identity* is omitted, 1.0 is used as the magnitude and the format of *float* is used. For example:

```
(float-sign -0.0) => -1.0
```

The function **float-radix** returns the radix used for the exponent in the format used for *float*. On the Explorer system, floating-point exponents are always powers of 2, so **float-radix** ignores its argument and always returns 2.

The function **float-digits** returns the number of significant bits of the mantissa in whatever the floating-point format is for *float*. For the field characteristics of floating-point numbers on the Explorer, see Table 3-1.

The function **float-precision** returns the number of radix digits in the mantissa of *float*. Since the radix is always 2 on the Explorer system, this function returns the number of significant bits.

Logical Operations on Numbers

3.10 The arguments to the following functions must be integers, which are treated as binary numbers in two's complement notation. Note that the examples for the following functions use octal numbers as arguments and that the returned value is octal. However, when you execute these examples on the Explorer system, the displayed values depend on whatever ***print-base*** is set to.

lognot *integer* *[c]* Function

This function returns the bitwise logical complement of *integer*. Note the following equivalence:

```
(lognot integer) <=> (logxor integer -1)
(logbitp j (lognot x)) <=> (not (logbitp j x))
```

Consider the following example:

```
(lognot #o3456) => #o-3457          ; Equivalent to 7774321 octal.
```

logior &rest *integers* *[c]* Function

This function returns the bitwise logical *inclusive or* of *integers*. If no arguments are given, **logior** returns 0, which is the identity for this operation. For example:

```
(logior #o4002 #o67) => #o4067
```

logxor &rest *integers* *[c]* Function

This function returns the bitwise logical *exclusive or* of *integers*. If no arguments are given, **logxor** returns 0, which is the identity for this operation. For example:

```
(logxor #o2531 #o7777) => #o5246
```

logand &rest *integers* *[c]* Function

This function returns the bitwise logical *and* of *integers*. If no arguments are given, **logand** returns -1, which is the identity for this operation. For example:

```
(logand #o3456 #o707) => #o406
(logand #o3456 #o-100) => #o3400
```

logeqv &rest *integers* *[c]* Function

This function returns the bitwise logical *equivalence* (also known as *exclusive nor*) of *integers*. This function returns -1 if the two argument bits are equal. This operation is associative. If no arguments are given, **logeqv** returns -1, which is the identity for this operation. Consider the following example:

```
(logeqv #o2531 #o7707) => #o-5237          ; Equivalent to 7772541 octal.
```

lognand *integer1 integer2* *[c]* Function

This function returns the bitwise logical *nand* of *integer1* and *integer2*. If either *integer1* or *integer2* is 0, **lognand** returns -1. Note the following equivalence:

```
(lognand integer1 integer2) <=> (lognot (logand integer1 integer2))
```

lognor *integer1 integer2* *[c]* Function

This function returns the bitwise logical *nor* of *integer1* and *integer2*. If both *integer1* and *integer2* are 0, **lognor** returns -1. Note the following equivalence:

```
(lognor integer1 integer2) <=> (lognot (logior integer1 integer2))
```

logandc1 *integer1 integer2* *[c]* Function

This function returns the bitwise logical *and* of *integer1*'s complement and *integer2*. Note the following equivalence:

```
(logandc1 integer1 integer2) <=> (logand (lognot integer1) integer2)
```

logandc2 *integer1 integer2* *[c]* Function

This function returns the bitwise logical *and* of *integer1* and the complement of *integer2*. Note the following equivalence:

```
(logandc2 integer1 integer2) <=> (logand integer1 (lognot integer2))
```

logorc1 *integer1 integer2* *[c]* Function

This function returns the bitwise logical *or* of *integer1*'s complement and *integer2*. Note the following equivalence:

```
(logorc1 integer1 integer2) <=> (logior (lognot integer1) integer2)
```

logorc2 *integer1 integer2* *[c]* Function

This function returns the bitwise logical *or* of *integer1* and the complement of *integer2*. Note the following equivalence:

```
(logorc2 integer1 integer2) <=> (logior integer1 (lognot integer2))
```

Table 3-4 summarizes the ten bitwise logical operations that can be performed on two integers.

Table 3-4 Bitwise Logical Operations on Two Integers

Function Name					Logical Operation
integer1	**0**	**0**	**1**	**1**	
integer2	**0**	**1**	**0**	**1**	
logand	0	0	0	1	And
logior	0	1	1	1	Inclusive or
logxor	0	1	1	0	Exclusive or
logeqv	1	0	0	1	Equivalence (exclusive nor)
lognand	1	1	1	0	Nand
lognor	1	0	0	0	Nor
logandc1	0	1	0	0	And complement of *integer1* with *integer2*
logandc2	0	0	1	0	And *integer1* with complement of *integer2*
logorc1	1	1	0	1	Or complement of *integer1* with *integer2*
logorc2	1	0	1	1	Or *integer1* with complement of *integer2*

boole *op intg1 intg2*	*[c]* Function
boole *op intg1* &rest *more-intg*	Function
boole-clr	*[c]* Constant
boole-set	*[c]* Constant
boole-1	*[c]* Constant
boole-2	*[c]* Constant
boole-c1	*[c]* Constant
boole-c2	*[c]* Constant
boole-and	*[c]* Constant
boole-ior	*[c]* Constant
boole-xor	*[c]* Constant
boole-eqv	*[c]* Constant
boole-nand	*[c]* Constant
boole-nor	*[c]* Constant
boole-andc1	*[c]* Constant
boole-andc2	*[c]* Constant
boole-orc1	*[c]* Constant
boole-orc2	*[c]* Constant

The **boole** function is the generalization of **logand**, **logior**, and **logxor**. This function returns the result of performing the logical operation *op* (which can be specified by one of the preceding constants) on *intg1* and *intg2*.

With two arguments, the result of **boole** is simply its second argument. At least two arguments are required.

Table 3-5 summarizes the Boolean logical operations that can be performed on two integers.

Table 3-5 Bitwise Boolean Operations on Two Integers

Function Name					Logical Operation
intg1	0	0	1	1	
intg2	0	1	0	1	
boole-clr	0	0	0	0	Always 0
boole-set	1	1	1	1	Always 1
boole-1	0	0	1	1	Returns *intg1*
boole-2	0	1	0	1	Returns *intg2*
boole-c1	1	1	0	0	Complement of *intg1*
boole-c2	1	0	1	0	Complement of *intg2*
boole-and	0	0	0	1	And
boole-ior	0	1	1	1	Inclusive or
boole-xor	0	1	1	0	Exclusive or
boole-eqv	1	0	0	1	Exclusive nor
boole-nand	1	1	1	0	Nand
boole-nor	1	0	0	0	Nor
boole-andc1	0	1	0	0	And the complement of *intg1* with *intg2*
boole-andc2	0	0	1	0	And *intg1* with the complement of *intg2*
boole-orc1	1	1	0	1	Or the complement of *intg1* with *intg2*
boole-orc2	1	0	1	1	Or *intg1* with the complement of *intg2*

If **boole** has more than three arguments, it is associated left to right (where *bl-cnst* is one of the **boole** constants previously listed) as follows:

```
(boole bl-cnst x y z) <=> (boole bl-cnst (boole bl-cnst x y) z)
```

The **boole** function can be useful when the logical operation is selected at run time. Also note that the Common Lisp primitive **boole** accepts only three arguments, whereas **boole** on the Explorer can accept more than three arguments and, thus, is an extension.

logtest *integer1 integer2* *[c]* Function

This function is a predicate that returns true if any of the bits designated by the 1-bits in *integer1* are 1-bits in *integer2*. Note the following equivalence:

```
(logtest integer1 integer2) <=> (not (zerop (logand integer1 integer2)))
```

logbitp *index integer* *[c]* Function

This function returns true if the bit *index* (in relation to the least significant bit in *integer*) is a 1. Note the following equivalence:

```
(logbitp index integer) <=> (ldb-test (byte index 1) integer)
```

Consider the following example:

```
(logbitp 1 7) => t                ; Or true
(logbitp 4 7) => nil              ; Or false
```

lsh *integer count* Function

This function logically shifts *integer* left by *count* bit positions or right by *count* bit positions if *count* is negative. Unused positions are filled by 0-bits that are shifted in (at either end). The arguments must be fixnums. Note that fixnums are only 25 bits wide and that **lsh** does not perform a circular shift. For example:

```
(lsh 4 1) => 8
(lsh #o14 -2) => 3
(lsh 1 25) => 0
(lsh 1 24) => -16777216
```

ash *integer count* *[c]* Function

This function arithmetically shifts *integer* left by *count* bits if *count* is positive, or right by *-count* bits if *count* is negative. Unused positions are filled by 0s from the right and by copies of the sign bit from the left. Thus, unlike **lsh**, the sign of the result is always the same as the sign of *integer*. If *integer* is a fixnum or a bignum, this is a shifting operation. If *integer* is a floating-point number, this function performs scaling (multiplication by a power of 2) rather than actually shifting any bits.

```
(ash 1 1) => 2
(ash 1 10) => 1024
(ash 2 -1) => 1
(ash 2 -2) => 0
(ash 1 23) => 8388608           ; A fixnum.
(ash 1 24) => 33554432          ; A bignum.
```

Notice that Common Lisp specifies only **ash** for integer arguments, whereas the Explorer system also allows the first argument to be a floating-point number.

rot *integer count* Function

This function returns *integer* rotated to the left by *count* bit positions if *count* is positive or 0, and rotated to the right if *count* is negative. On the Explorer system, the rotation considers *integer* as a 25-bit number, and both arguments must be fixnums. This function does not operate on bignums. This function is best avoided because it is highly implementation-dependent. Consider the following examples:

```
(rot 1 2) => 4
(rot 1 -2) => #o40000000
(rot -1 7) => -1
(rot 15 25) => 15
```

logcount *integer* *[c]* Function

This function returns the number of 1-bits in *integer* if this argument is positive or returns the number of 0-bits in *integer* if this argument is negative. (A negative integer logically contains an infinite number of 1-bits because the sign bit extends to the left as many places as necessary.) For example:

```
(logcount #o15) => 3
(logcount #o-15) => 2
(logcount 13) => 3
(logcount -13) => 2
(logcount 30) => 4
(logcount -30) => 4
```

integer-length *integer* *[c]* Function

This function returns the minimum number of bits (excluding the sign) needed to represent *integer* in two's complement notation. For example:

```
(integer-length 0) => 0
(integer-length 7) => 3
(integer-length 8) => 4
(integer-length -7) => 3
(integer-length -8) => 3
(integer-length -9) => 4
```

haulong *integer* Function

This function returns the number of significant bits in the absolute value of *integer*, which can be a fixnum or a bignum. The sign of *integer* is ignored. The result is the least integer strictly greater than the base-2 logarithm of the absolute value of *integer*. Note the following equivalence:

```
(haulong x) <=> (integer-length (abs x))
```

haipart *integer n* Function

This function returns the *n* highest bits of the absolute value of *integer*, or the *n* lowest bits if *n* is negative. The *integer* argument can be a fixnum or a bignum; its sign is ignored.

Byte Manipulation Functions

3.11 The following functions manipulate bytes through the use of byte specifiers. In the following descriptions, a *byte* is any bit string, not just those with eight bits. A *byte specifier* denotes a particular byte position within an integer and a field width. Byte specifiers are normally created by the **byte** function.

On the Explorer system, byte specifiers are integers whose lowest six bits represent the size of the byte and whose higher bits (usually 6) represent the position of the byte within the integer (beginning at 0 and counting from the right in bits). Because of this arrangement, byte specifiers are easier to understand when displayed in octal. The maximum size of a byte is 63 bits.

byte *size position* *[c]* Function
byte-size *byte-spec* *[c]* Function
byte-position *byte-spec* *[c]* Function

The **byte** function returns a byte specifier for the byte of *size* bits, positioned to exclude the number of least significant bits specified by *position*. This byte specifier can be passed as the first argument to **ldb**, **dpb**, and **mask-field**. The **byte-size** function returns the size of the byte specified by *byte-spec*, and the **byte-position** function returns the position of the byte specified by *byte-spec*.

For the following example, suppose that you want to specify an eight-bit byte that starts in bit position 24:

```
(byte 8 24) => 1544 = #o3010
(byte-size 1544) => 8
(byte-position #o3010) => 24
```

For these three functions, note the following equivalence:

(**byte** (**byte-size** *byte-spec*) (**byte-position** *byte-spec*)) <=> *byte-spec*

ldb *byte-spec integer* *[c]* Function

This function (which stands for *load byte*) extracts a byte (specified by the argument *byte-spec*) from the argument *integer*. The result is returned as a positive integer. For example:

```
(ldb (byte 6 3) #o4567) => #o56
```

Note that `ldb`'s returned value is right-justifed; that is, there are no zeros to the right of `56` as there are in the subsequent example for **mask-field**.

You can use **setf** with **ldb** to change a byte in the integer located at *place* if the *integer* argument to **ldb** meets the requirements for **setf**. In effect, this operation is analogous to invoking **dpb** with the returned value being stored at *place*.

signed-ldb *byte-spec integer* *[c]* Function

This function is like **ldb** except that the top bit of the extracted byte is taken to be a sign bit.

ldb-test *byte-spec integer* *[c]* Function

This function is a predicate that returns true if any of the bits designated by *byte-spec* is a 1 in the argument *integer*; that is, **ldb-test** returns true if the designated field specified by *byte-spec* is nonzero. Note the following equivalence:

```
(ldb-test a-byte-spec n) <=> (not (zerop (ldb a-byte-spec n)))
```

mask-field *byte-spec integer* *[c]* Function

This function is like the **ldb** function; however, the specified byte of *integer* is positioned in the same byte-spec of the returned value. The returned value is 0 outside of that byte. The *integer* argument must be an integer. For example:

```
(mask-field (byte 6 3) #o4567) => #o560
```

You can use **setf** with **mask-field** to change a byte in the integer located at *place* if the *integer* argument to **mask-field** meets the requirements for **setf**. In effect, this operation is analogous to invoking **deposit-field** with the returned value being stored at *place*.

dpb *newbyte byte-spec integer* *[c]* Function

This function (whose name stands for *deposit byte*) returns a number that is composed by substituting the bits of *newbyte* for the *byte-spec* bits of *integer*, which must be an integer. The *newbyte* argument is interpreted as being right-justified, as if it were the result of **ldb**. If *newbyte* is a larger number than the size of *byte-spec*, then the most-significant bits of *newbyte* are ignored. The *integer* argument can be a fixnum or a bignum. For example:

```
(dpb #o23 (byte 6 3) #o4567) => #o4237
```

deposit-field *newbyte byte-spec integer* *[c]* Function

This function returns an integer that is composed by substituting the *byte-spec* bits of *newbyte* for the *byte-spec* bits of *integer*, which must be an integer. This function is similar to **dpb**, but *newbyte* is not taken to be right-justified. For example:

```
(deposit-field #o230 (byte 6 3) #o4567) => #o4237
```

Random Numbers

3.12 The following functions are associated with the Common Lisp pseudo random number generator. The *random-state* arguments to these functions refer to objects of type **random-state** which contain the state of the pseudo random number generator between calls to **random**.

random *number* &optional *random-state* — *[c]* Function
random &optional *number random-state* — Function

This function returns a randomly generated number. If *number* is specified, the random number is of the same type as *number* (floating if *number* is floating, and fixed if *number* is an integer), is nonnegative, and is less than *number*.

If *number* is omitted, the result is a randomly chosen fixnum, with all fixnums being equally probable.

If *random-state* is specified, it is used and updated in generating the random number. Otherwise, the default random state is used (and is created if it does not already exist). The algorithm is executed inside a **without-interrupts** form so that two processes can use the same random state without colliding.

random-state — *[c]* Variable

This variable contains the default random state used by **random**.

When **random** is invoked, it causes a side effect on the value of this variable. You can bind it to a different random generator state object and restore the old state.

make-random-state &optional *random-state* — *[c]* Function

This function creates and returns a new random state object. If *random-state* is **nil**, the new random state is a copy of ***random-state***. If *random-state* is a random-state object, the new random-state object is a copy of this argument. If *random-state* is **t**, the new random state is actually initialized randomly (based on the current time value).

NOTE: In the case of **make-random-state**, the value **t** has a specific meaning; thus, substituting any non-**nil** value instead of **t** for *random-state* does not produce the result described above.

---

random-state-p *object* — *[c]* Function

This predicate returns true if *object* is a random-state object; otherwise, it returns **nil**.

Number Type Functions

3.13 The following functions either test numbers to determine if they are of a particular type or coerce an object into a number of a particular type.

numberp *object* *[c]* Function

This predicate returns true if *object* is a number and otherwise returns **nil**. For example:

```
(numberp -5) => true
(numberp 0.00000000001) => true
(numberp 'a) => nil
```

integerp *object* *[c]* Function

This predicate returns true if *object* is an integer and otherwise returns **nil**. For example:

```
(integerp 5) => true
(integerp 0.00000000001) => nil
(integerp 'a) => nil
```

fixnump *object* Function

This function returns true if *object* is a fixnum.

bigp *object* Function

This function returns true if *object* is a bignum.

rationalp *object* *[c]* Function

This predicate returns true if *object* is a rational number (a ratio or an integer) and otherwise returns **nil**.

floatp *object* *[c]* Function

This predicate returns true if *object* is a floating-point number and otherwise returns **nil**. For example:

```
(floatp 5) => nil
(floatp 1.0s12) => true
(floatp 'a) => nil
```

complexp *object* *[c]* Function

This predicate returns true if *object* is a complex number and otherwise returns **nil**. For example:

```
(complexp 5) => nil
(complexp #c(5 2)) => true
(complexp (sqrt -4)) => true
```

realp *object* Function

This predicate returns true if *object* has a value of type **real**. Any fixnum, bignum, floating-point number (of any size), or rational number satisfies this predicate. Otherwise, **realp** returns **nil**. Note the following equivalence:

```
(realp x) <=> (and (numberp x) (not (complexp x)))
```

zerop *number* *[c]* Function

This predicate returns true if *number* equals 0. The *number* argument must be a number. Note the following equivalence:

```
(zerop x) <=> (= x 0)
```

Consider the following example:

```
(zerop 5) => nil
(zerop 0) => true
(zerop 0.0) => true
(zerop #c(0.0 0.0)) => true
(zerop 'a) => ERROR
```

plusp *number* *[c]* Function

This predicate returns true if *number* is a number greater than 0. Otherwise, it returns **nil**. If *number* is complex or is not a number, **plusp** signals an error. Note the following equivalence:

```
(plusp x) <=> (> x 0)
```

Consider the following examples:

```
(plusp 5) => true
(plusp 0) => nil
(plusp -2.5s3) => nil
(plusp 'a) => ERROR
```

minusp *number* *[c]* Function

This predicate returns true if *number* is a number less than 0. Otherwise, it returns **nil**. The form `(minusp -0.0)` is always false. If *number* is complex or is not a number, **minusp** signals an error. Note the following equivalence:

```
(minusp x) <=> (< x 0)
```

Consider the following examples:

```
(minusp 5) => nil
(minusp 0) => nil
(minusp -0.0) => nil
(minusp -2.5s3) => true
(minusp 'a) => ERROR
```

oddp *integer* *[c]* Function

This predicate returns true if *integer* is *odd* and otherwise returns **nil**. The *integer* argument must be an integer.

evenp *integer* *[c]* Function

This predicate returns true if *integer* is *even* and otherwise returns **nil**. The *integer* argument must be an integer.

CHARACTERS

4

Character Definitions

4.1 The **character** data type has two primary subtypes: **string-char** and **standard-char**, with **standard-char** being a subtype of **string-char**. This section defines those functions that deal with **standard-char** objects, whereas Section 8, Strings, deals with functions that handle **string-char** objects.

Common Lisp allows some latitude in the implementation of characters. One of the side effects of this latitude, however, is that portable programs cannot assume that characters are conventional data objects. One resulting anomaly is that the **eq** function may not reliably operate on characters. For example:

```
(let  ((x char)
       (y char))
   (eq x y))
```

If `char` is a character object, this expression may not return true. The implementation of characters on the Explorer system returns true for this case, but, for portable Common Lisp programs, use the **eql** function to test for identity of character objects.

An object of type **character** is defined as having three attributes:

- The code attribute is a number ranging from 0 to one less than the value of **char-code-limit**. On the Explorer, **char-code-limit** is set to 256.
- The font attribute is a number that indicates a particular font. This number ranges from 0 (the default) to one less than the value of **char-font-limit**. On the Explorer system, **char-font-limit** is set to 256. The mapping of a font attribute to a specific font is maintained by the window system. See the *Explorer Window System Reference* manual for details.
- A bit attribute allows you to represent modified characters. Although you can treat this attribute as a bit mask, it can also be used as a number between 0 (the default) and one less than the value of **char-bits-limit**. All alphabetic characters have a bit attribute of zero. On the Explorer, six bit attributes correspond to the four character-modifying keys and the mouse buttons. These bits are named Control, Meta, Super, Hyper, Mouse, and Keypad. Mouse and Keypad are not part of the Common Lisp definition.

To create a character object, you can use one of the support functions or use a #\ prefix for a given character. This prefix indicates a character object rather than a symbol and suppresses the lowercase to uppercase mapping that is otherwise performed by the Lisp Reader. For example:

```
(defparameter vowels '(#\a #\A #\e #\E #\i #\I #\o #\O #\u #\U))
```

This example creates a variable named `vowels`, which consists of a list of the vowel characters. The default font number is taken to be 0, and no bit attributes are set.

Nongraphics character objects are those that do not have a normal printed representation or that have at least one bit attribute set. These characters can be referred to by using the #\ prefix and the character name. For example, `#\tab` identifies the tab character. Refer to Table 4-1 for the complete list of character names. By convention, `#\space` is considered to be graphic.

You can specify a particular font for a character object by placing a font number between the # and the \. For instance, `#3\a` identifies a lowercase letter a in font number 3. Again, remember that on the Explorer this number is used as an index into the font map maintained by the window in which the character is displayed. Since the font map can vary from window to window, the appearance of the character may also vary.

By convention, Common Lisp specifies that graphics characters in font 0 are of constant width, which can be handy when you are printing out tables. On the other hand, nongraphics characters in font 0 and all characters in other fonts should be assumed to be of variable width (possibly 0).

Except for the mouse bit, characters with bit attributes set can be represented using #\ and the name or initial of the bit attributes, each separated by a dash. For example:

```
#\CTRL-META-4          <=>   #\C-M-4
#\HYPER-SPACE          <=>   #\H-SPACE
#\SUPER-*              <=>   #\S-*
#\KEYPAD-SPACE         <=>   #\K-SPACE
```

If you want to name lowercase characters with control bits, you need to protect the lowercase letter from the Reader; the initial backslash only affects the reading of of the control-bit name. For example, `#\ctrl-\x` is a character with the control bit set and whose character code is equal to a lowercase x. For example:

```
(= (char-code #\ctrl-\x)
   (char-code #\x))
=> T
```

However, if, when you are typing a character or printing with the **format** directive -c, any of the control bits are set, the logic of the SHIFT key is reversed with regard to the character code. For example:

```
(format nil "-c" #\ctrl-\x)
=> "ctrl-sh-X"                         ; Identifies a shifted X.

(char= #\ctrl-\x
       (send *terminal-io* :tyi))
=> T                                   ; If you press CTRL-SHIFT-x.
```

Mouse characters are represented as follows:

Character Objects	Mouse Clicks
`#\mouse-L-1`	Click the left button once.
`#\mouse-L-2`	Click the left button twice.
`#\mouse-L-3`	Click the left button three times.
`#\mouse-M-1`	Click the middle button once.
`#\mouse-M-2`	Click the middle button twice.
`#\mouse-M-3`	Click the middle button three times.
`#\mouse-R-1`	Click the right button once.
`#\mouse-R-2`	Click the right button twice.
`#\mouse-R-3`	Click the right button three times.

Keypad buttons (that is, the block of keycaps at the lower righthand side of the keyboard) generate character codes that are the same as the typewriter keycaps, except that the keypad bit may or may not be set. Whether this bit is set is a characteristic of the window in which the process is running. If you want the bit set, you must specify **:keypad-enable t** as an option when you create the window. Only the following characters can have the keypad bit set:

0 1 2 3 4 5 6 7 8 9 = + - , . #\space #\tab #\return

Note that when the keypad bit is set for these characters, they become nongraphic and do not print as simple characters. See the *Explorer Window System Reference* manual for details.

Standard and Nonstandard Characters

4.2 Common Lisp programs that are intended to be portable should contain only characters from the standard character set. The Common Lisp character set contains a space character (`#\space`), a newline character (`#\newline`), and the following 94 characters:

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^
_ a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ `
```

The following characters are supported on the Explorer but are considered only semistandard by Common Lisp:

`#\backspace`   **`#\tab`**   **`#\linefeed`**   **`#\page`**   **`#\return`**   **`#\rubout`**

Common Lisp defines the `#\newline` character as being the only legal line delimiting character. For the Explorer system, the `#\return` character is the same as `#\newline`, and both correspond to the RETURN key.

The Explorer system supports the standard Common Lisp character set, the International Standards Organization (ISO) character set, and additional characters supported by most Lisp machines. Table 4-1 lists the complete Explorer character set in sequential order; the decimal and octal codes represent the corresponding code attributes for each character. Note that the displayed representation for any character may vary with different fonts. The printed characters listed in this table are from the cptfont font and should be judged as the standard for other fonts.

Whenever the character name is not given in Table 4-1, the character object can be referenced as #*character*. Thus, the exclamation mark is named #\!. Depending on the application, some keystrokes are interpreted as commands. For example, to print the down arrow character, you press CTRL-Q and the ↓ key.

Table 4-1 Explorer Character Set

Decimal	Octal	Print	Keystroke	Character Name(s)
0	000	·	SYMBOL-SHIFT-"	#\center-dot
1*	001	↓	↓	#\down-arrow, #\hand-down
2	002	α	SYMBOL-SHIFT-A	#\alpha
3	003	β	SYMBOL-SHIFT-B	#\beta
4	004	∧	SYMBOL-Q	#\and-sign
5	005	¬	SYMBOL-SHIFT-{	#\not-sign
6	006	ϵ	SYMBOL-SHIFT-E	#\epsilon
7	007	π	SYMBOL-SHIFT-P	#\pi
8	010	λ	SYMBOL-SHIFT-L	#\lambda
9	011	γ	SYMBOL-SHIFT-G	#\gamma
10	012	δ	SYMBOL-SHIFT-D	#\delta
11*	013	↑	↑	#\up-arrow, #\hand-up
12	014	±	SYMBOL-SHIFT-:	#\plus-minus
13	015	⊕	SYMBOL-SHIFT-<	#\circle-plus
14	016	∞	SYMBOL-I	#\infinity
15	017	∂	SYMBOL-P	#\partial-delta
16	020	⊂	SYMBOL-T	#\left-horseshoe
17	021	⊃	SYMBOL-Y	#\right-horseshoe
18	022	∩	SYMBOL-E	#\up-horseshoe
19	023	∪	SYMBOL-R	#\down-horseshoe
20	024	∀	SYMBOL-U	#\universal-quantifier
21	025	∃	SYMBOL-O	#\existential-quantifier
22	026	⊗	SYMBOL-SHIFT->	#\circle-x, #\circle-cross
23	027	⇆	SYMBOL-L	#\double-arrow
24*	030	←	←	#\left-arrow, #\hand-left
25*	031	→	→	#\right-arrow, #\hand-right
26	032	≠	SYMBOL-C	#\not-equal, #\not-equals
27*	033	◇	ESCAPE	#\escape, #\esc, #\altmode, #\alt
28	034	≤	SYMBOL-N	#\less-or-equal
29	035	≥	SYMBOL-M	#\greater-or-equal
30	036	≡	SYMBOL-B	#\equivalence
31	037	∨	SYMBOL-W	#\or-sign, #\or
32	040		SPACE	#\space, #\sp
33	041	!		
34	042	"		
35	043	#		
36	044	$		
37	045	%		
38	046	&		
39	047	'		

*Characters marked with an asterisk can be entered as shown here only when preceded by the CTRL-Q key sequence. Thus, to enter ↓ you must press CTRL-Q ↓.

Table 4-1 Explorer Character Set (Continued)

Decimal	Octal	Print	Keystroke	Character Name(s)
40	050	(		
41	051	)		
42	052	*		
43	053	+		
44	054	,		
45	055	-		
46	056	.		
47	057	/		
48	060	0		
49	061	1		
50	062	2		
51	063	3		
52	064	4		
53	065	5		
54	066	6		
55	067	7		
56	070	8		
57	071	9		
58	072	:		
59	073	;		
60	074	<		
61	075	=		
62	076	>		
63	077	?		
64	100	@		
65	101	A		
66	102	B		
67	103	C		
68	104	D		
69	105	E		
70	106	F		
71	107	G		
72	110	H		
73	111	I		
74	112	J		
75	113	K		
76	114	L		
77	115	M		
78	116	N		
79	117	O		
80	120	P		
81	121	Q		
82	122	R		
83	123	S		
84	124	T		
85	125	U		
86	126	V		
87	127	W		
88	130	X		
89	131	Y		
90	132	Z		
91	133	[		
92	134	\		

Table 4-1 Explorer Character Set (Continued)

Decimal	Octal	Print	Keystroke	Character Name(s)
93	135	]		
94	136	^		
95	137	_		
96	140	`		
97	141	a		
98	142	b		
99	143	c		
100	144	d		
101	145	e		
102	146	f		
103	147	g		
104	150	h		
105	151	i		
106	152	j		
107	153	k		
108	154	l		
109	155	m		
110	156	n		
111	157	o		
112	160	p		
113	161	q		
114	162	r		
115	163	s		
116	164	t		
117	165	u		
118	166	v		
119	167	w		
120	170	x		
121	171	y		
122	172	z		
123	173	{		
124	174	|		
125	175	}		
126	176	~		
127	177	∫	SYMBOL-SHIFT-?	#\integral
128	200	⟨NULL⟩	<no keystroke>	#\null, #\null-character
129	201	⟨BREAK⟩	BREAK	#\break, #\brk
130*	202	⟨CLEAR INPUT⟩	CLEAR-INPUT	#\clear-input, #\clear
131	203	⟨CALL⟩	<no keystroke>	#\call
132	204	⟨TERM⟩	TERM	#\term, #\terminal
133	205	⟨MACRO⟩	<no keystroke>	#\macro, #\back-next
134*	206	⟨HELP⟩	HELP	#\help
135*	207	⟨RUBOUT⟩	RUBOUT	#\rubout
136	210	⟨OVERSTRIKE⟩	<no keystroke>	#\overstrike,#\backspace, #\bs
137	211	⟨TAB⟩	TAB	#\tab
138	212	⟨LINE FEED⟩	LINEFEED	#\linefeed, #\line, #\line-feed, #\lf

*Characters marked with an asterisk can be entered as shown here only when preceded by the CTRL-Q key sequence. Thus, to enter ↓ you must press CTRL-Q ↓.

Table 4-1 Explorer Character Set (Continued)

Decimal	Octal	Print	Keystroke	Character Name(s)
139	213	DELETE	<no keystroke>	#\delete, #\vt
140*	214	CLEAR SCREEN	CLEAR-SCREEN	#\clear-screen,#\page, #\form,#\ff, #\refresh
141	215	RETURN	RETURN	#\return,#\cr,#\newline
142	216	QUOTE	<no keystroke>	#\quote
143	217	HOLD OUTPUT	<no keystroke>	#\hold-output
144	220	STOP OUTPUT	<no keystroke>	#\stop-output
145	221	ABORT	ABORT	#\abort
146*	222	RESUME	RESUME	#\resume
147*	223	STATUS	STATUS	#\status
148*	224	END	END	#\end
149	225	F1	F1	#\f1, #\function-1, #\roman-I
150	226	F2	F2	#\f2, #\function-2, #\roman-II
151	227	F3	F3	#\f3, #\function-3, #\roman-III
152	230	F4	F4	#\f4, #\function-4, #\roman-IV
153	231	LEFT	LEFT	#\left
154	232	MIDDLE	MIDDLE	#\middle
155	233	RIGHT	RIGHT	#\right
156	234	CENTER	CENTER	#\center, #\center-arrow
157	235	SYSTEM	SYSTEM	#\system
158	236	NETWORK	NETWORK	#\network
159	237	UNDO	UNDO	#\undo
160	240			#\no-break-space
161	241	¡	SYMBOL-1	#\inverted-exclamation mark
162	242	¢	SYMBOL-2	#\american-cent-sign
163	243	£	SYMBOL-3	#\british-pound-sign
164	244	¤	SYMBOL-4	#\currency-sign
165	245	¥	SYMBOL-5	#\japanese-yen-sign
166	246	¦	SYMBOL-‘	#\broken-bar
167	247	§	SYMBOL-6	#\section-symbol
168	250	¨	SYMBOL-’	#\diaresis
169	251	©	SYMBOL-7	#\copyright-sign
170	252	ª	SYMBOL-8	#\feminine-ordinal-indicator
171	253	«	SYMBOL-,	#\angle-quotation-left
172	254	¬		
173**	255	-	SYMBOL--	#\soft-hyphen
174	256	®	SYMBOL-0	#\registered-trademark
175	257	¯	SYMBOL--	#\macron
176**	260	°	SYMBOL-.	#\degree-sign
177	261	±		
178**	262	²	SYMBOL-2	#\superscript-2

*Characters marked with an asterisk can be entered as shown here only when preceded by the CTRL-Q key sequence. Thus, to enter ↓ you must press CTRL-Q ↓.

**This keystroke is defined with a number or symbol from the numeric keypad. You cannot use the number or symbol on the typewriter keypad for this keystroke.

Table 4-1 Explorer Character Set (Continued)

Decimal	Octal	Print	Keystroke	Character Name(s)
179**	263	³	SYMBOL-3	#\superscript-3
180	264	´	SYMBOL-SHIFT-ESCAPE	#\acute-accent
181	265	µ	SYMBOL-SHIFT-M	#\greek-mu,#\mu
182	266	¶	SYMBOL-(	#\paragraph-symbol
183	267	•		
184	270	¸	SYMBOL-;	#\cedilla
185**	271	¹	SYMBOL-1	#\superscript-1
186	272	º	SYMBOL-9	#\masculine-ordinal-indicator
187	273	»	SYMBOL-.	#\angle-quotation-right
188**	274	¼	SYMBOL-4	#\fraction-1/4
189**	275	½	SYMBOL-5	#\fraction-1/2
190**	276	¾	SYMBOL-6	#\fraction-3/4
191	277	¿	SYMBOL-/	#\inverted-question-mark
192	300	À	SYMBOL-!	
193	301	Á	SYMBOL-@	
194	302	Â	SYMBOL-#	
195	303	Ã	SYMBOL-$	
196	304	Ä	SYMBOL-%	
197	305	Å	SYMBOL-^	
198	306	Æ	SYMBOL-&	
199	307	Ç	SYMBOL-*	
200	310	È	SYMBOL-SHIFT-9	
201	311	É	SYMBOL-SHIFT-0	
202	312	Ê	SYMBOL-SHIFT--	
203	313	Ë	SYMBOL-+	
204	314	Ì	SYMBOL-}	
205**	315	Í	SYMBOL-SHIFT-=	
206**	316	Î	SYMBOL-SHIFT-+	
207	317	Ï	SYMBOL-SHIFT-Q	
208	320	Ð	SYMBOL-SHIFT-W	
209	321	Ñ	SYMBOL-SHIFT-R	
210	322	Ò	SYMBOL-SHIFT-T	
211	323	Ó	SYMBOL-SHIFT-Y	
212	324	Ô	SYMBOL-SHIFT-U	
213	325	Õ	SYMBOL-SHIFT-I	
214	326	Ö	SYMBOL-SHIFT-O	
215**	327	×	SYMBOL-+	#\multiplication-sign
216	330	Ø	SYMBOL-[	
217	331	Ù	SYMBOL-]	
218	332	Ú	SYMBOL-SHIFT-\	
219**	333	Û	SYMBOL-SHIFT-7	
220**	334	Ü	SYMBOL-SHIFT-8	
221**	335	Ý	SYMBOL-SHIFT--	

**This keystroke is defined with a number or symbol from the numeric keypad. You cannot use the number or symbol on the typewriter keypad for this keystroke.

Table 4-1 Explorer Character Set (Continued)

Decimal	Octal	Print	Keystroke	Character Name(s)
222	336	Þ	SYMBOL-SHIFT-S	
223**	337	ß	SYMBOL-8	#\eszet
224	340	à	SYMBOL-SHIFT-F	
225	341	á	SYMBOL-SHIFT-H	
226	342	â	SYMBOL-SHIFT-J	
227	343	ã	SYMBOL-SHIFT-K	
228**	344	ä	SYMBOL-SHIFT-4	
229**	345	å	SYMBOL-SHIFT-5	
230**	346	æ	SYMBOL-SHIFT-6	
231	347	ç	SYMBOL-SHIFT-Z	
232	350	è	SYMBOL-SHIFT-X	
233	351	é	SYMBOL-SHIFT-C	
234	352	ê	SYMBOL-SHIFT-V	
235	353	ë	SYMBOL-SHIFT-N	
236**	354	ì	SYMBOL-7	
237**	355	í	SYMBOL-9	
238**	356	î	SYMBOL-SHIFT-1	
239**	357	ï	SYMBOL-SHIFT-2	
240**	360	ð	SYMBOL-SHIFT-3	
241**	361	ñ	SYMBOL-SHIFT-0	
242**	362	ò	SYMBOL-SHIFT-.	
243	363	ó	SYMBOL-A	
244	364	ô	SYMBOL-D	
245	365	õ	SYMBOL-F	
246	366	ö	SYMBOL-Z	
247**	367	÷	SYMBOL-=	#\division-sign
248	370	ø	SYMBOL-X	
249	371	ù	SYMBOL-V	
250**	372	ú	SYMBOL-,	
251**	373	û	SYMBOL-SHIFT-,	
252**	374	ü	SYMBOL-SHIFT-<space>	
253**	375	ý	SYMBOL-SHIFT-<tab>	
254	376	þ	SYMBOL-S	
255**	377	ÿ	SYMBOL-SHIFT-9	

**This keystroke is defined with a number or symbol from the numeric keypad. You cannot use the number or symbol on the typewriter keypad for this keystroke.

Character Attributes

4.3 The following constants are associated with a character's attributes.

char-code-limit *[c]* Constant

The value of this constant is one more than the maximum code attribute of any character. On the Explorer system, this value is currently 256.

char-font-limit *[c]* Constant

The value of this constant is one more than the maximum font attribute value of any character. On the Explorer, this value is currently 256.

char-bits-limit *[c]* Constant

The value of this constant is one more than the maximum modifier bit attribute value of any character. On the Explorer, currently, this value is 64. Thus, there are six bit attributes: the Control, Meta, Super, Hyper, Mouse, and Keypad bits.

Character Construction and Attribute Retrieval

4.4 The following functions are used for constructing characters and retrieving information about character attributes.

char-code *char* *[c]* Function

This function returns the code attribute value of *char*. This returned attribute is a nonnegative integer less than **char-code-limit**. Constants have been defined for the individual control bits; see paragraph 4.6, Character Control Bit Functions. The *char* argument must be a character object. The code attribute values for the Explorer character set are shown in Table 4-1. Consider the following example:

```
(char-code #\b) => 98  ;98 decimal is 142 octal.
```

char-bits *char* *[c]* Function

This function returns the bit attribute value of *char*. This returned attribute is a nonnegative integer less than **char-bits-limit**. The *char* argument must be a character object.

char-font *char* *[c]* Function

This function returns the font attribute value of *char*. This returned attribute is a nonnegative integer less than **char-font-limit**. The *char* argument must be a character object.

char-mouse-button *char* Function

This function returns 0, 1, or 2 if *char* is a left, middle, or right mouse button character button, respectively.

char-mouse-clicks *char* Function

This function returns 0, 1, or 2 if *char* is a single, double, or triple mouse click, respectively (that is, it returns the number of clicks minus 1). For example:

```
(char-mouse-clicks #\mouse-m-2) => 1
```

code-char *code* &optional *bits font* *[c]* Function

This function returns a character object whose attributes are specified by the arguments *code*, *bits*, and *font*, which must be nonnegative integers less than the values of **char-code-limit**, **char-bits-limit**, and **char-font-limit**, respectively. If the arguments do not comply with the code, bits, and font limitations, then **nil** is returned. Any combination of these attributes is valid if the arguments are valid individually, except for the keypad bit, whose code value must correspond to one of the keypad buttons. For example:

```
(code-char #o141) => #\a

(code-char 32 char-control-bit) => #\c-SPACE
```

make-char *char* &optional *bits font* *[c]* Function

This function returns the character specified by *char* with the bit and font attributes set by the *bits* and *font* arguments, which must be nonnegative integers less than the values of **char-bits-limit** and **char-font-limit**, respectively. If the *bits* and *font* arguments do not comply with the character attribute limitations of the system, then **nil** is returned. This function differs from **code-char** only in that the first argument for **make-char** is a character object instead of an integer.

char-name *char* *[c]* Function

This function returns the standard name (or one of the standard names) for the argument *char* (which must be a character object), or **nil** if there is none. The name is returned as a string. For example:

```
(char-name #\space) => "SPACE"
```

As this example shows, any character denoted by a name (rather than by a letter or a digit) is specified as #*character-name*. (See paragraph 4.1, Character Definitions.)

If the *char* argument has nonzero modifier bits, the returned value is **nil**. Compound names such as **Control-X** are not constructed by this function.

name-char *name* *[c]* Function

This function returns (as a character object) the character for which *name* is a name, or returns **nil** if *name* is not a recognized character name. The *name* argument is coerced to a string and compared to the known character names using **string-equal**. Compound names such as **Control-X** are not recognized. Consider the following example:

```
(name-char "SPACE") => #\space
```

The **read** function uses this function to process the **#** construct when a character name is encountered.

Character Conversion

4.5 The following functions are used for character conversion operations.

char-upcase *char* *[c]* Function

If the *char* argument is a lowercase alphabetic character, then this function returns a character object whose character code attribute is mapped to the corresponding uppercase alphabetic character. The bit and font attributes of the argument stay the same.

If the *char* argument is not alphabetic, **char-upcase** returns *char* unchanged. Note that a character with a nonzero bit attribute is not considered alphabetic. (See **digit-char-p** and **graphic-char-p.**)

char-downcase *char* *[c]* Function

If the *char* argument is an uppercase alphabetic character, then this function returns a character object whose character code attribute is mapped to the corresponding lowercase alphabetic character. The bit and font attributes of the argument stay the same.

If the *char* argument is not alphabetic, **char-downcase** returns *char* unchanged. Note that a character with a nonzero bit attribute is not considered alphabetic. (See **digit-char-p** and **graphic-char-p.**)

digit-char *magnitude* &optional *radix font* *[c]* Function

This function returns a character object that is the digit with the specified magnitude and in the specified radix and font. However, if there is no suitable character that has the magnitude specified by *magnitude* in the specified *radix* (which defaults to 10), the returned value is **nil**. If the returned value is alphabetic (that is, if *magnitude* is greater than 9), its returned character is uppercase. For example:

```
(digit-char 5) => #\5
(digit-char 10) => nil
(digit-char 10 16) => #\A        ; Not #\a.
```

The **digit-char** function does not have an argument for specifying the bit attribute of the character to be returned because digits (which are graphics characters) always have a bit attribute of 0. (See **digit-char-p** and **graphic-char-p.**)

char-int *char* *[c]* Function

This function converts *char*, a character object, to the integer that represents the same character. This function is the inverse of **int-char** and is used mainly for hashing characters.

As an Explorer extension, this function can also be given a fixnum as an argument, in which case the fixnum is returned.

If *char* has both a font attribute and a bit attribute of 0, then the value returned by **char-int** is the same as that returned by **char-code**. If these attributes are not 0, then they are added to the character's code attribute value after being shifted left past the most-significant bit of the code attribute value. See the *Explorer System Software Design Notes* for more details on the format of this number.

int-char *integer* *[c]* Function

This function converts *integer*, regarded as representing a character, to a character object. If a character object is given as an argument, it is returned unchanged. If *integer* does not correspond to a character object, then **int-char** returns **nil**. This function is the inverse of **char-int**.

Character Control Bit Functions

4.6 The following constants and functions are used for operations involving the Control, Meta, Super, Hyper, Mouse, and Keypad bit attributes.

char-control-bit *[c]* Constant
char-meta-bit *[c]* Constant
char-super-bit *[c]* Constant
char-hyper-bit *[c]* Constant
char-mouse-bit Constant
char-keypad-bit Constant

These constants have the values 1, 2, 4, 8, 16, and 32, respectively. They give numerical meaning to the bit configuration within the bit attribute of a character object. Thus, the following form evaluates to true if *char* is a character whose bit attribute has the Meta bit set.

```
(logtest char-meta-bit (char-bits char))
```

char-bit *char name* *[c]* Function

This function returns true if *char* has its bit attribute set to the indicated *name*. The valid values for *name* are **:control**, **:meta**, **:super**, **:hyper**, **:mouse**, and **:keypad**. Any other value produces an error. For example:

```
(char-bit #\HYPER-Y :HYPER) => true

(char-bit #\HYPER-SUPER-Y :HYPER) => true
```

The **setf** macro can be used with **char-bit**, provided that the *char* argument is a form acceptable to **setf**, to alter the bit attribute of the character stored in the location specified by *char*. Note also that **char-bit** is an access function that returns a Boolean value. This combination of **setf** and **char-bit** is similar to performing a **set-char-bit** operation. For example:

```
(setq x #\a)
(char-bit x :control) => nil
(setf (char-bit x :control) t) => t
(char-bit x :control) => true
(setf (char-bit x :control) nil) => nil
(char-bit x :control) => nil
```

set-char-bit *char name set-flag* *[c]* Function

This function returns a character that is equal to *char* and whose bit attribute is set to the bit name specified by *name* if *set-flag* is non-**nil**. Otherwise, it returns a character whose code attribute is equal to *char* but whose bit attribute is not set. The *name* argument must be one of the following: **:control**, **:meta**, **:super**, **:hyper**, **:mouse**, or **:keypad**. For example:

```
(set-char-bit #\X :meta t) => #\META-X
(set-char-bit #\X :meta nil) => #\X
(set-char-bit #\META-X :meta t) => #\META-X
(set-char-bit #\META-X :meta nil) => #\X
```

For more information about mouse click characters, see the *Explorer Window System Reference* manual.

Character Type Functions

4.7 The following functions either test characters to determine if they are of a particular type or coerce an object into a character.

characterp *object* *[c]* Function

This predicate returns true if *object* is a character and otherwise returns **nil**. For example:

```
(characterp #\x) => true
(characterp #\7) => true
(characterp #\') => true
(characterp 'x) => nil
(characterp 7) => nil
```

standard-char-p *char* *[c]* Function

This predicate returns true if *char* is a standard Common Lisp character of type **standard-char**. This type includes 94 printing characters and the blank characters #\space and #\newline (see paragraph 4.2, Standard and Nonstandard Characters), and it requires the bit and font attributes to be zero.

graphic-char-p *char* *[c]* Function

This predicate returns true if *char* is a graphic character, that is, one with a printed shape. The character #\space is a graphics character; #\return, #\end, and #\abort are not. A character whose bit attribute is nonzero is never a graphics character.

Ordinary output to windows prints graphics characters using the current font. Nongraphics characters are printed using lozenges unless they have special formatting meanings (as #\return does). See Table 4-1.

string-char-p *char* *[c]* Function

This predicate returns true if *char* is a character that can be stored in a Common Lisp string, and otherwise returns **nil**. For any *char*, if `(standard-char-p char)` returns true, then so does `(string-char-p char)`. On the Explorer system, **string-char-p** returns true for all characters with bit and font attributes of zero.

alpha-char-p *char* *[c]* Function

This predicate returns true if *char* is a letter of the alphabet whose bit attribute is 0. Otherwise, this predicate returns **nil**.

upper-case-p *char* *[c]* Function

This predicate returns true if *char* is an uppercase letter with a bit attribute of 0.

lower-case-p *char* *[c]* Function

This predicate returns true if *char* is a lowercase letter with a bit attribute of 0.

both-case-p *char* *[c]* Function

This predicate returns true if *char* is a character that has distinct uppercase and lowercase forms. On the Explorer, it returns the same value as **alpha-char-p**, except for the #\eszet character.

digit-char-p *char* &optional *radix* *[c]* Function

This predicate returns the magnitude of *char* if *char* is a digit available in the specified radix. Otherwise, it returns **nil**. The *radix* argument defaults to 10. If the bit attribute of *char* is nonzero, **digit-char-p** always returns **nil**. For example:

```
(digit-char-p #\8 8) => nil
(digit-char-p #\8 9) => 8
(digit-char-p #\F 16.) => 15
(digit-char-p #\c-8 any-old-thing) ;Because the control bit is set,
=> nil                             ;the radix is insignificant.
```

alphanumericp *char* *[c]* Function

This predicate returns true if *char* has a bit attribute of 0 and if *char* returns true either for **alpha-char-p** or **digit-char-p** (radix 10).

character *object* *[c]* Function

This function coerces *object* into a character and returns the character as a character object.

Character Comparisons

4.8 The following functions are character predicates that perform character comparisons. They operate in a manner similar to the number comparison functions. Note that Common Lisp specifies that all uppercase letters will collate correctly, that all lowercase letters will collate correctly, and that digits 0-9 will collate correctly. However, it does not specify how a mixture of uppercase and lowercase letters will collate. Thus, the letter *A* may be greater than the letter *a*, or the letter *a* may be greater than the letter *A*.

char= *char* &rest *more-characters*	*[c]* Function
char/= *char* &rest *more-characters*	*[c]* Function
char< *char* &rest *more-characters*	*[c]* Function
char> *char* &rest *more-characters*	*[c]* Function
char<= *char* &rest *more-characters*	*[c]* Function
char>= *char* &rest *more-characters*	*[c]* Function

These are the functions for comparing characters that consider the code, font, and bit attribute in the comparison. On the Explorer system, the numeric functions are called =, >, and so on.

The ordering of the characters is based on the integer value of the code attribute of all characters. This order is shown in Table 4-1. Consider the following example:

```
(char= #\b #\b) => true
(char= #\b #\x)) => nil
(char= #\b #\B) => nil
(char= #\b #\b #\b #\b) => true
(char= #\b #\b #\x #\b) => nil

(char/= #\b #\b) => nil
(char/= #\b #\B) => true
(char/= #\b #\b #\b #\b) => nil
(char/= #\b #\b #\x #\b) => nil
(char/= #\b #\z #\x #\c) => true

(char< #\b #\x) => true
(char< #\b #\b) => nil
(char< #\b #\f #\x #\y) => true
(char< #\b #\f #\f #\x)  => nil

(char<= #\b #\x) => true
(char<= #\b #\b) => true
(char<= #\b #\f #\x #\y) => true
(char<= #\b #\f #\f #\x) => true

(char> #\f #\e) => true
(char> #\e #\d #\c #\b) => true
(char> #\e #\e #\d #\a) => nil
(char> #\Z #\b) => nil

(char>= #\f #\e) => true
(char>= #\e #\d #\c #\b) => true
(char>= #\e #\e #\d #\a) => true
```

You can also use the predicates **eql** and **equal** for comparing characters for equality, but you should not use **eq** because its behavior for characters may differ in various Common Lisp implementations.

char-equal *character* &rest *more-characters* *[c]* Function
char-not-equal *character* &rest *more-characters* *[c]* Function
char-lessp *character* &rest *more-characters* *[c]* Function
char-not-lessp *character* &rest *more-characters* *[c]* Function
char-greaterp *character* &rest *more-characters* *[c]* Function
char-not-greaterp *character* &rest *more-characters* *[c]* Function

These functions are for comparison of characters, ignoring the character's case as well as its font attribute. These functions are called by many of the string functions. On the Explorer, the arguments can be integers or character objects. For example:

```
(char-equal #\x #\x) => true
(char-equal #\X #\x) => true
(char-equal #\X #\META-X) => true
```

5

PACKAGES

Package Definitions

5.1 The *package* system is a facility in the Lisp environment that provides a mapping between print names and symbols. The chief purpose of the package system is to provide a relatively isolated *namespace*, or set of accessible symbols, for each separate application that is loaded. This arrangement is necessary because it is possible that two applications will want to use the same symbol name for different purposes. As a result, when the symbols of two different applications are partitioned into packages, the complete Lisp environment becomes more structured.

The package system is primarily made up of package objects, which provide logical groupings of symbols, and relationships established between package objects and the symbols they contain. Those symbols grouped into a particular package are said to be *owned* by that package. A symbol that is owned by a particular package is said to be *interned* in that package. A symbol that is not owned by any package, such as those generated by **gensym**, is said to be *uninterned*.

In general, the term *interned* means that a particular Lisp object is uniquely identifiable in some context. With regard to packages, the objects in question are symbols and the context is a namespace. When a symbol is interned, it becomes uniquely identifiable by the symbol name within a namespace context.

Because there is no owning package for uninterned symbols, an uninterned symbol cannot exist in any namespace; consequently, there is no way to map a name reference to the symbol object using the package system. This section deals mainly with interned symbols and how the package system establishes the mappings of name to object.

Although the notion of symbols being grouped into packages is fairly straightforward, the nature of the relationships that can exist between packages and the way in which they establish a namespace can be quite complex. The following paragraphs explain the mechanics of these relationships.

Overview of a Symbol Namespace

5.1.1 The data type **package** defines an object that defines its own namespace. The Lisp system requires that one package be known as the *current package*, and thus the namespace that it defines becomes the current namespace environment. Using a syntactic package prefix (which is explained later), you can explicitly refer to any symbol in the Lisp environment. Symbols that do not have an explicit package prefix must be *accessible* in the current namespace environment.

By definition, referring to a symbol without explicitly using a package qualifier means that the symbol is in the current namespace and is accessible. Implicitly, if a symbol of that name is not found, one is created in the current package and that symbol's home package (the symbol-package cell) is set to the current package.

When you define an application, you should make some symbols publicly accessible, such as the main entry point or user-settable parameters. Other symbols, whose purpose is meant to be unadvertised, should not be readily accessible. This capability is provided by allowing each package to have two distinct classes of symbols: *internal* and *external*. When symbols are created, they are internal by default. They can be made external by using the **export** function.

To access a symbol in another namespace, you must use a package qualifier. If `flag` is an external symbol in the package `MY-PKG`, then the reference to this symbol is `my-pkg:flag`. If `flag` were an internal symbol, then the more cumbersome double-colon syntax, `my-pkg::flag`, must be used. If you find yourself using numerous references to internal symbols, there is probably a design problem in your program.

All of the symbols defined by Common Lisp are exported from the LISP package. By default, all of the new symbols typed in the Lisp Listener are interned as internal symbols in the USER package.

Symbols interned in the KEYWORD package are automatically exported, as well as having their value bound to themselves. Note that, syntactically, the printed or typed reference to a symbol in the KEYWORD package need not specify the package name in the qualifier prefix because the empty string is a nickname for the KEYWORD package. Also note that the **symbol-value** of a keyword refers to the symbol itself.

```
(eq keyword:flag :flag) => true  ; The KEYWORD package has a nickname
                                 ; of "".
(eq ':flag :flag) => true        ; Symbol-value of flag is 'flag.
(symbol-name :flag) => "FLAG"    ; The colon is not part of the print name.
```

Usually, one application depends on another application or subsystem, and therefore the symbols in one package are dependent on the symbols in the other package. This relationship, in which one package *uses* the other package, allows code in one package to *inherit* the externally declared symbols of the other package. An inherited symbol is treated as an internal symbol in the inheriting package, but the external symbols of the other package are not affected by this inheritance. An inherited symbol is treated as if it were an internal symbol in the inheriting package. Gaining access to a symbol via inheritance does not change the symbol's status as an external symbol in the used package, nor does it change the symbol's home package. When a symbol is inherited in a namespace it can be referred to without a package prefix. The term *non-external symbols* includes these inherited symbols plus the native internal symbols in the current package. In the most basic case, almost all applications depend on the symbols defined in the LISP package. For example, the USER package uses the LISP package. At run time, an application can explicitly control on which package it depends.

Another way that a package can provide access to a symbol in a different package is by *importing* that symbol. Each imported symbol must be explicitly identified in a call to the **import** function (as opposed to inheriting all the external symbols of a used package). A key difference between an imported symbol and an inherited symbol is that imported symbols are actually added as internal symbols to the package that imports them and are said to be *present* in that package. This presence is significant because when the symbol mapping takes place, it is resolved by finding the symbol in the current package, as opposed to finding an inherited symbol from a used package. Gaining access to a symbol via importing does not change the symbol's home package.

When a package *uses* the symbols in another package, it inherits only the external symbols in the other package. This usage does not cause any new symbols to be present in any package. Note that access of symbols is not transitive across packages. For example:

1. Assume that `symbol-x` is an exported symbol of `package-x`.
2. If `package-y` uses `package-x`, then you can reference `symbol-x` from `package-y`.
3. If `package-z` uses `package-y`, `package-z` cannot automatically access the same `symbol-x`.

Importing, on the other hand, does support transitive access of symbols. For example:

1. Assume that `symbol-x` is in `package-x`.
2. If `package-y` imports `package-x:symbol-x`, then you can reference `symbol-x` from `package-y`.
3. If `package-z` imports `package-y:symbol-x`, then you can reference the same `symbol-x` from `package-y` and `package-z`.

This scheme of referencing is considered safe because the user must explicitly list those symbols to be imported.

Using these techniques, a package can configure a namespace that is suitable for the intended application.

Consistency Rules

5.1.2 The definable interrelationships between one package and another can become quite complex. The following three rules are the basic precepts to which all functions that deal with packages must adhere. A clear understanding of these rules will help you learn how the package system works.

The following rules hold true as long as the current package is not changed:

- Read-read consistency — Reading the same print name always maps to the same (eq) symbol.
- Print-read consistency — An interned symbol always prints as a sequence of characters such that when that sequence is read back in, it maps to the same (eq) symbol.
- Print-print consistency — If two interned symbols are not eq, then their printed representations are a different sequence of characters.

These rules ensure that your program will be able to generate reproducible results.

It is not unusual that a program may need to change the current package and thus potentially cause some anomalies. Once the new package is made current, the consistency rules apply to references in that namespace. However, during the transition between packages, one of the rules may be violated. Changing back to the original package should resume the previous mapping.

However, some functions can cause these rules to be violated more permanently. These functions are mildly dangerous because they can alter the name-to-symbol mapping for a particular namespace. Once this operation is performed, it cannot be undone to return to the previous mappings, nor is any trace kept to to explain how the mappings were changed. The following list contains the dangerous functions:

unintern
unuse-package
unexport
shadowing-import
shadow

Package Names

5.1.3 When a package is created, a primary name is specified along with an optional list of nicknames. By convention, the primary name should be long and self-explanatory, whereas the nicknames should be short mnemonics. In practice, the short nicknames are used to save keystrokes whenever package qualifiers are needed.

Packages are referred to in two different ways: as the package object itself (when it is passed as an argument to a function) or as a package qualifier to a symbol name. For functions that take package objects as arguments, the package name can usually be passed instead of the package object itself. In this case, the name can be specified as a string or a symbol. If the name is a symbol, then the print name of the symbol is used. If a package name is identified by using a string, the case of the characters is important, so you should probably use uppercase characters.

When the Lisp Reader encounters a symbol that includes a package prefix qualifier, it maps lowercase letters to uppercase just as it does for unqualified symbol names. The Reader attempts to match the qualifier name to an existing package name in a case-sensitive manner. If lowercase letters become part of a package (nick)name and references are made using this package qualifier, then the lowercase letters must be protected. For instance, if there is a symbol named `FLAG` in a package called `"My-Pkg"`, then the reference should be written in either of the following ways:

```
M\y-P\k\g:FLAG
```

```
|My-Pkg|:FLAG
```

Also note that the package name is parsed separately from the symbol name. Thus, the following two symbol names are not the same:

```
(eq |My-Pkg|:FLAG |My-Pkg:FLAG|) => nil
```

The second argument to `eq` refers to a symbol in the current package whose print name is `"My-Pkg:FLAG"`.

Translating Strings to Symbols

5.1.4 Lisp code is executed with one specific package designated as the current package. Thus, the namespace that the current package defines is the set of symbols that are accessible without package qualifications. Three classes of symbols may be parsed by the Reader. If a symbol name is parsed without a package qualifier, the Reader first checks to see if the named symbol is accessible in the current namespace. The system first searches the current package and then the used packages. Note that the order in which the used packages are searched is not significant because no name conflicts can occur. The avoidance of name conflicts is guaranteed by all the functions that modify the namespace. If a corresponding symbol is found, then that symbol is returned. If a corresponding symbol is not found, then a new symbol is created and interned as an internal symbol in the current package.

If a package qualifier is used, the symbol's name is looked up in the specified package, not the current package. If the qualified name contains a single colon, Common Lisp specifies that an external symbol with that name must be accessible in the specified package. If a double colon is used, then the effect is the same as if the current package were changed to the specified package while the symbol mapping took place. If the package does not exist or if no appropriate symbol can be found, an error is signaled.

On the Explorer system, the Lisp Reader's handling of qualified symbol names is extended. Qualified names are still first looked up in the specified package and then in any package used by the specified package. If the package does not exist, an error is signaled, but in this case, if no appropriate symbol can be found, a new symbol is created and interned as an internal symbol in the specified package.

If the Reader parses a #:*<name>* symbol, then the Reader makes an uninterned symbol whose print name is "*<name>*". Because the symbol is not interned, the package cell is set to **nil**, indicating that the symbol does not have a home package.

Symbols are printed in a way that maintains the consistency rules. Uninterned symbols are printed as #:*<name>*. Interned symbols that are accessible in the current package are printed without a package qualifier. If the symbols are not accessible in the current namespace, then they are printed with one colon if they are external in their home package or two colons if they are not external in their home package.

Importing and Exporting Symbols

5.1.5 If a symbol is to be imported, it must be present as an external symbol in the package from which it is to be imported. By default, an imported symbol is made present in the current package as an internal symbol. If the same (**eq**) symbol is already present and external, then the symbol remains external. If a different symbol with the same name was previously accessible in the importing package, then a name conflict occurs.

If you want the imported symbol to shadow the currently accessible symbol, you should use the **shadowing-import** function. When a symbol with the same name is already interned in the importing package, then it is uninterned and the new symbol is added as an internal symbol.

A package can also gain access to a symbol in another package through use of the **use-package** function. This function causes the current package to inherit all of the external symbols of the package being used. These inherited symbols are accessible in the inheriting package but are not present in the inheriting package. There is no way to inherit internal symbols from another package.

If a symbol is to be exported from a package, it must first be accessible. If the symbol is already external, then exporting has no effect. If the symbol is currently an internal symbol, its status is simply changed to external. If the symbol is inherited and non-external, it is first imported to make it present and is then made external.

Note that a symbol can be present in several packages and that it can be marked external or internal in each package independently. Thus, it is the symbol's presence in a particular package that is external or not, rather than the symbol itself. The **export** function makes symbols external in whichever package you specify; if the same symbols are present in any other package, their status as external or internal in the other package is not affected.

Name Conflicts and Shadowing

5.1.6 When a namespace has its mapping scheme changed in some way, a name conflict can occur. Specifically, a name conflict means that a symbol name has at least two corresponding symbols (which are not **eq**) and that there is no meaningful way to consistently choose one symbol over the others, such as is provided with **shadowing-import**.

Functions that modify the namespace mapping of a symbol are required to check for name conflicts before completing execution. When such a name conflict is detected, an error is signaled. These conflicts can be resolved in the error handler by selecting the appropriate symbol to which the name should be mapped.

The **use-package** function checks each of the external symbols in the used package for name conflicts with the accessible symbols in the current namespace.

The **import** function checks to see that the symbol being added is not already accessible. If it is, a name conflict results even if the accessible symbol was originally made accessible by the **shadowing-import** function. This conflict occurs because the name has been explicitly called out for two different purposes.

The **export** function checks to see that the symbol being made external does not conflict with symbols that are accessible in other namespaces that are to inherit symbols from this package.

The **unintern** function checks to see if the symbol being removed is a shadowing symbol. If it is, then a further check is made to verify that there is no conflict between inherited symbols from its various used packages.

A name conflict can be resolved by using **shadowing-import, shadow, unintern,** or **unexport** on one of the conflicting symbols. If you abort from the error handler during a name conflict, the original symbol remains accessible.

Shadowing must be done before programs are loaded into the package, because if the programs are loaded without shadowing first, they contain pointers to the undesired inherited symbol. Merely shadowing the symbol at this point does not alter those pointers; only reloading the program and rebuilding its data structures from scratch can do that. If it is necessary to refer to a shadowed symbol, it can be done using a package prefix.

However, shadowing can be used to reject inheritance of any symbol. Shadowing is the primary means of resolving *name conflicts* in which an inherited symbol matches a symbol directly present in an inheriting package. The **shadowing-import** function is the primary means of resolving name conflicts in which multiple symbols with the same name are available (due to inheritance) in one package.

The conflict is resolved—always in advance—by placing the preferred choice of symbol in the package directly (if it is not already present), and marking it as a *shadowing symbol*. This can be done with the function **shadow** when the preferred choice is already present or with **shadowing-import** when it is not. (Actually, you can proceed from the error and specify a resolution, but this works by shadowing and retrying. From the point of view of the retried operation, the resolution has been done in advance.)

Major Built-In Packages

5.1.7 The following major packages are built into the Explorer system:

- LISP — All the standard functions and variables of Common Lisp are present as external symbols in this package.
- TICL — All Explorer extensions to the Common Lisp language and user interface are present as external symbols in this package. Additionally, functions and variables used in many program-development utilities reside as external symbols in the TICL package. Implementation-dependent extensions are for the most part relegated to the SYSTEM package. However, this distinction is not rigidly maintained because certain useful extensions and utilities may be system-dependent in various ways.
- SYSTEM — Symbols shared among various Explorer-specific system programs are included as external symbols in this package. Low-level, system-dependent routines not typically mentioned in documentation are included as internal symbols within this package.
- ZLC — All functions and variables that are obsolete for Common Lisp or that are Explorer extensions to Common Lisp are present as external symbols in this package. All Zetalisp-specific symbols that are incompatible with their Common Lisp namesakes are present as internal symbols in this package. When Zetalisp mode is turned on, the Lisp Reader and evaluator access the incompatible Zetalisp symbols instead of the ones defined in the LISP or TICL packages.
- GLOBAL — This package is provided to be a look-alike, as much as possible, of the GLOBAL package in earlier versions of the Explorer software. It exports all the ZLC symbols (including the incompatible and internal ones) and all of the LISP and TICL symbols, except those that conflict with the ZLC internal symbols.

- USER — The USER package is the default package for input typed by the user. Initially, it contains no symbols. This package uses the LISP, TICL, and ZLC packages.
- KEYWORD — This package contains as external symbols all the keywords used by built-in or user-defined functions. Because the package-printing prefix for the KEYWORD package in the empty string, printed symbols representations that begin with a colon refer to symbols in this package. All such symbols are treated as constants that refer to themselves.

Nicknames are used to preserve compatibility with software prior to Release 3.0. The following nicknames are supported.

Package	Nicknames
LISP	CLI, COMMON-LISP-INCOMPATIBLE
TICL	EECL
SYSTEM	SYS, SI, SYSTEM-INTERNALS
ZLC	none
GLOBAL	ZETALISP, ZL, ZETALISP-GLOBAL
KEYWORD	"" (The empty string)

Defining Packages

5.2 The following macro and associated functions are used for defining packages.

defpackage *name* &key **:nicknames :size :use :prefix-name :export :import :shadow :shadowing-import :auto-export-p** Macro

This macro defines a package specified by *name,* which should be a string or symbol. None of the arguments to **defpackage** are evaluated. The keyword arguments are passed in a keyword association-list format, but eventually are parsed and sent to **make-package** as normal keyword arguments.

If a package already exists with the name specified in *name*, it is modified insofar as this is possible to correspond to the new definition.

The following are the possible options and their meanings:

(**:nicknames** {*name-string*}*) — A list of nicknames for the new package. The nicknames should be specified as strings. If a package of the same name already exists, an error is signaled.

(**:size** *number-of-symbols*) — The number of symbols that the new package is initially made large enough to hold before a rehash is needed (interned symbols are kept in hash tables).

(**:use** {*package-name*}*) — A list of packages or names for packages that the new package should inherit from, or a single name or package. It defaults to the LISP and TICL packages.

(**:prefix-name** *package-name*) — Specifies the name to use for printing package prefixes that refer to this package. It must be equal to either the package name or one of the nicknames.

(**:export** *args*)
(**:import** *args*)
(**:shadow** *args*)
(**:shadowing-import** *args*) — If any of these arguments are non-nil, they are passed to the function of the same name to operate on the package. Note that some of these functions accept strings, and others accept symbols. See the explanations of the individual functions for details.

NOTE: The only exception to this is the **:export** option, which accepts strings. If the arguments were symbols instead of strings, the reader would intern those arguments into the current package before **defpackage** started. This would cause those symbols to be imported to the new package and then exported. Since this is probably not what you want, you should always specify strings to the **:export** option.

Consider the following example:

```
(defpackage some-package
  (:nicknames "SP")
  (:export "BEGIN")
  (:import another-package: help)
  (:shadow "LOGIN" "LOGOUT"))
```

This form creates/alters SOME-PACKAGE, and interns four symbols:

- BEGIN — A newly created symbol that has the home package SP and is marked as exported.
- `help` — An internal symbol in SP that has the home package ANOTHER-PACKAGE.
- LOGIN and LOGOUT — Internal symbols in SP which have the home package SP. Presumably, these local symbols are masking the symbols of the same name in the LISP package, which SP uses by default.

  Note that when the **:export**, **:import**, **:shadow**, and **:shadowing-import** functions are called, the new package has already been created and is set as the current package while these functions are called. You could accomplish the same thing by calling **export**, **import**, **shadow**, or **shadowing-import** yourself.

:auto-export-p — If this option is non-**nil**, all symbols interned in the new package are automatically exported.

For example, the EH system package could have been defined this way:

```
(defpackage eh
  (:size 1200)
  (:use "LISP" "TICL" "SYSTEM")
  (:nicknames "DBG" "DEBUGGER")
  (:shadow "ARG"))
```

This form performs the following operations for the EH package:

- Creates 1200 symbol entries in the package
- Creates the nicknames DBG and DEBUGGER
- Uses the LISP, TICL, and SYSTEM packages
- Contains a symbol named arg that is not the same as the **arg** in the LISP package

It is usually best to put the package definition in a separate file, which should be loaded into the USER package. (It cannot be loaded into the package it is defining, and no other package has any reason to be preferred.) Often the files to be loaded into the package belong to one system or to a few systems; therefore, it is often convenient to put the system definitions in the same file.

A package can also be defined by the package attribute in a file's attribute line. Normally, this attribute specifies in which (existing) package to load, compile, or edit the file. But suppose the attribute value is a list, as in the following:

```
;-*-Package: (FOO :size 300); ...-*-
```

In this case, loading, compiling, or editing the file automatically creates the FOO package with the specified options (exactly like the **make-package** options). No **defpackage** is needed. It is wise to use this feature only when the package is used for just a single file. For programs containing multiple files, it is better to make a system for them and then put a **defpackage** near the **defsystem**. (See Section 23, Maintaining Large Systems.)

make-package *name* &key **:nicknames :use** *[c]* Function
make-package *name* &key **:nicknames :size :use :prefix-name :export :shadow :import :shadowing-import :auto-export-p** Function

This function creates and returns a new package with the name specified by *name*, which can be a string or symbol. The functionality of the keywords is the same as for **defpackage**, although, for **make-package** they are true keywords; the **defpackage** arguments are actually in an association-list format. Only **:nicknames** and **:use** are defined by Common Lisp. The other options are Explorer extensions.

in-package *name* &key **:nicknames :use** *[c]* Function
in-package *name* &key **:nicknames :size :use :prefix-name :export :shadow :import :shadowing-import :auto-export-p** Function

This function creates a package named by *name*, if it does not exist, with the nicknames specified by **:nicknames** and the used packages specified by **:use**, or it modifies an existing package named by *name* to have the specified nicknames and used packages. Finally, the current package is set to this package. This binding remains in effect until changed by the user or until the current package reverts to its previous value at the completion of a load operation. The *name* argument can be a string, symbol, or package object.

The keywords **:nicknames** and **:use** are defined by Common Lisp. All other options are Explorer extensions that are passed to **make-package** if the package specified by *name* does not exist. Otherwise, the package specified by *name* is modified insofar as possible to correspond to the extended keyword options.

delete-package *package* Function

This function uninterns all the symbols in the package specified by *package*, invokes **unuse-package** on all the packages *package* is currently using, and deletes *package*.

kill-package *package* Function

This function kills the package specified or named by *package*. It is removed from the list that is searched when package names are looked up.

Setting the Current Package

5.3 The name of the current package is always displayed in the middle of the status line with a colon following it. This package name describes the process that the status line in general is describing, normally the process of the selected window. No matter how the current package is changed, the status line eventually shows this change (at one-second intervals). Thus, while a file is being loaded, the status line displays that file's package; the status line displays the package of the selected buffer in the editor.

The following forms are used for setting the current package.

package *[c]* Variable

The value of this variable is the current package. The **intern** function searches this package if it is not given a second argument. Many other functions for operating on packages also use this variable as the default.

Setting or binding the variable changes the current package.

NOTE: Do not set this variable to a value that is not a package object! It must be set to a package at all times.

Each process or stack group can have its own setting for the current package by binding ***package*** with **let**. The actual current package at any time is the value bound by the process that is running. The bindings of another process are irrelevant until the process runs.

When a file is loaded, ***package*** is bound to the package named in the file's attribute line (see paragraph 1.3.3, Using the Two Modes From Zmacs) when the file has an attribute line. The Chaosnet program file has `Package:CHAOS;` in the attribute line, and therefore its symbols are looked up in the CHAOS package. An object file has an encoded representation of the attribute line for the source file; this representation looks different from the actual attribute line, but it serves the same purpose.

The current package is also relevant when you type Lisp expressions on the keyboard; it controls the reading of the symbols that you type. Initially, it is the USER package. You can select a different package using **in-package**, or even by setting ***package***. If you are working with the Chaosnet program, it might be useful to type `(in-package 'CHAOS)` so that your symbols are found in the CHAOS package by default. The Lisp Listener loop binds ***package*** so that **in-package** in one Lisp Listener does not affect others or any other processes whatever.

The Zmacs editor records the correct package for each buffer; the package is determined from the file's attribute line. This package is used whenever expressions are read from the buffer. So if you edit the definition of the Chaosnet **get-packet** and recompile it, the new definition is read in the CHAOS package. The current buffer's package is also used for all expressions or symbols typed by the user. Thus, if you press META-. and type `allocate-pbuf` while looking at the Chaosnet program, you get the definition of the `allocate-pbuf` function in the CHAOS package.

pkg-bind *package* {*body*}* Macro

With this macro, the forms of the *body* are evaluated sequentially with the variable ***package*** bound to the package named by *package*. The argument *package* can be a package object or a package name. For example:

```
(pkg-bind "ZWEI"
   (some-zwei-function an-arg))
```

pkg-goto *package* &optional *globally-p* Function

This function sets ***package*** to *package* if *package* is suitable for this argument. The *package* argument can be specified as a package object or the name of one. If *globally-p* is non-**nil**, then this function also calls **pkg-goto-globally**.

pkg-goto-globally *package* Function

This function sets the global binding of ***package*** to *package*. An error is signaled if *package* is not suitable.

The variable ***package*** also has a global binding, which is in effect in any process or stack group that does not rebind the variable. New processes that do bind ***package*** generally use the global binding to initialize their own bindings, invoking `(let((*package* *package*))...)`. Thus, it can be useful to set the global binding. But you cannot do this with **setf** or **in-package** from a Lisp Listener, or in a file, because doing so sets the local binding of ***package*** instead. Therefore, you must use **pkg-goto-globally**.

Interning Symbols

5.4 The following forms are associated with interning symbols.

intern *string* &optional *package* *[c]* Function
intern *string-or-symbol* &optional *package* Function

This function searches *package* for a symbol whose print name is equal to *string* (or *string-or-symbol*). If *package* is not specified, the current package is searched instead. If such a symbol is found, it is returned as the first value of **intern**. Otherwise, each package used by *package* (or the current package) is searched for an external symbol with print name *string* (or *string-or-symbol*) until either such a symbol is found or all used packages have been searched. If the symbol is found, it is returned as the first value of **intern**. If it is not found, a new symbol is created with print name *string* (or *string-or-symbol*) and a home package of *package* (or the current package). In Common Lisp, the first argument to **intern** must be a string. On the Explorer system, the first argument can also be a symbol, in which case the print name of the symbol is used in the search.

The **intern** function also returns two additional values. The second value indicates whether an existing symbol was found and how. This second value is one of the following:

- **:internal** — A symbol was found directly present in *package*, and it was internal in *package*.
- **:external** — A symbol was found directly present in *pkg*, and it was external in *package*.
- **:inherited** — A symbol was found by inheritance from a package used by *package*. You can deduce that the symbol is external in that package.
- **nil** — A new symbol was created.

On the Explorer system, a third value is returned by **intern** indicating in which package the symbol found or created is present directly. This value is different from *package* if and only if the second value is **:inherited.**

Note that **intern** is sensitive to case; that is, it considers two character strings different even if they vary only by characters being uppercase or lowercase (unlike most string comparisons elsewhere in the Explorer system). Symbols are converted to uppercase when you type them because the Lisp Reader converts the case of characters in symbols; the characters are converted to uppercase before **intern** is ever called.

unintern *symbol* &optional *package* *[c]* Function

This function removes *symbol* from *package* and, if *package* is the home package for *symbol*, sets the package cell to **nil**. The *package* argument defaults to the current package. The *symbol* argument is also removed from *package*'s shadowing-symbols list if it is present there. If *symbol* is not present in *package*, no action is taken. The **unintern** function returns **t** if a symbol is actually removed; otherwise, it returns **nil.**

If a shadowing symbol is removed, several distinct symbols with the same name may become accessible in *package*. If this happens, an error is signaled. On proceeding, you can either leave *symbol* in *package* or choose which conflicting symbol should remain accessible. The chosen symbol is then made present in *package* as a shadowing symbol.

intern-local *string-or-symbol* &optional *package* Function

This function is like **intern**, but it ignores inheritance. If a symbol whose name matches *string-or-symbol* is present directly in *package*, it is returned; otherwise, *string-or-symbol* (if it is a symbol) or a new symbol (if *string-or-symbol* is a string) is placed directly in *package*. The *package* argument defaults to the current package.

The **intern-local** function returns second and third values with the same meaning as those of **intern**. However, the second value can never be **:inherited**, and the third value is always *package*.

find-symbol *string* &optional *package* *[c]* Function
find-symbol *string-or-symbol* &optional *package* Function

This function is like **intern** but never creates a new symbol or modifies *package*. If no existing symbol is found, **nil** is returned for all three values. If a symbol with the specified name is found in *package*, it is returned as the first value of **find-symbol**. Two additional values with the same meaning as those specified for **intern** are also returned. In Common Lisp, the first argument to **find-symbol** should be a string. On the Explorer system, the first argument can also be a symbol, in which case the print name of the symbol is used.

Inheritance Between Packages

5.5 The following functions are used to set up and control package inheritance.

import *symbols* &optional *package* *[c]* Function

This function interns each member of *symbols* in *package*. The *symbols* argument can also be an individual symbol. The *package* argument defaults to the current package. For each symbol in *symbols*, the following is done:

- If an identical (**eq**) symbol is present in *package*, nothing is done.
- If the symbol is accessible by inheritance in *package*, it is interned in *package*.
- If the symbol is not accessible in *package* and there is no distinct symbol of the same name already accessible in *package*, it is interned in *package*.
- Otherwise, a name conflict is detected, and an error is signaled. On proceeding, you can choose which conflicting symbol to make accessible. If the symbol being imported is chosen, it is then made present in *package* as a shadowing symbol.

use-package *packages* &optional *in-package* *[c]* Function

This function makes *in-package* inherit symbols from *packages*, which should be a single package or a list of packages.

The **use-package** function can cause name conflicts in two ways. First, if any of the *packages* has an external symbol whose name matches a symbol directly present in *in-package*, an error is signaled. On proceeding, you can either make a shadowing symbol out of the symbol already present in *in-package*, or you can choose to unintern the conflicting symbol from *in-package*. Resolving the conflict in the latter way is dangerous if the symbol to be uninterned is an external symbol, because other packages may rely on its presence in *in-package*.

In the second kind of name conflict, as with **unintern** and **export**, several distinct symbols with the same name may become accessible in *in-package*. If this happens, an error is signaled. On proceeding, you can choose which conflicting symbol should remain accessible. The chosen symbol is then made present in *in-package* as a shadowing symbol.

unuse-package *packages* &optional *in-package* *[c]* Function

This function makes *in-package* cease to inherit symbols from *packages*, which should be a single package or a list of packages. No name conflicts are possible because no new symbols are made accessible.

package-use-list *package* *[c]* Function

This function returns the list of packages used by *package*.

package-used-by-list *package* *[c]* Function

This function returns the list of packages that use *package*.

export *symbols* &optional *package* *[c]* Function

This function makes *symbols* external in *package*. The *symbols* argument should be a symbol or a list of symbols. If the symbols are not already present in *package*, they are imported first. The *package* argument defaults to the current package.

The **export** function can cause name conflicts in two ways. First, if the symbol being exported matches a symbol already present in a package that would inherit the newly exported symbol, an error is signaled. On proceeding, you can either unintern the symbol present in the inheriting package or choose to make it a shadowing symbol.

In the second kind of name conflict, as with **unintern** and **use-package**, several distinct symbols with the same name may become accessible in an inheriting package. If this happens, an error is signaled. On proceeding, you can choose which conflicting symbol should remain accessible in the inheriting package. The chosen symbol is then made present in that package as a shadowing symbol.

unexport *symbols* &optional *package* *[c]* Function

This function makes *symbols* not external in *package*. No name conflicts are possible because no new symbols are made accessible. However, an error is signaled if any of the *symbols* are not directly present in *package* or if *package* is used by other packages.

package-auto-export-p *package* Function

This function returns true if *package* automatically exports all symbols inserted in it.

package-external-symbols *package* Function

This function returns a list of all the external symbols of *package*.

Functions Associated With Shadowing and Name Conflicts

5.6 The following forms are associated with shadowing and name conflicts.

shadow *names* &optional *package* *[c]* Function

This function makes sure that shadowing symbols with the specified names exist in *package*. The *names* argument is either a string or symbol or a list of such. If symbols are supplied, their print names are used. Each name specified is handled independently as follows.

If there is a symbol of that name directly present in *package*, it is marked as a shadowing symbol to avoid any problems with name conflicts. Otherwise, a new symbol of that name is created and interned in *package* and is marked as a shadowing symbol.

shadowing-import *symbols* &optional *package* *[c]* Function

This function interns the specified *symbols* in *package* and marks them as shadowing symbols. The *symbols* argument must be a list of symbols or a single symbol; strings are not allowed.

Each symbol specified is placed directly into *package*, after uninterning any symbol with the same name already interned in *package*.

The **shadowing-import** function is primarily useful for choosing one of several conflicting external symbols that are present in packages to be used.

Once a package has a shadowing symbol named *foo* in it, any other potentially conflicting external symbol named *foo* can come and go in the inherited packages with no effect. It is, therefore, possible to use another package containing another *foo*, or to export the *foo* in one of the used packages, without causing an error.

package-shadowing-symbols *package* *[c]* Function

This function returns the list of shadowing symbols of *package*. Each of these is a symbol present directly in *package*. When a symbol is present directly in more than one package, it can be a shadowing symbol in one and not in another.

Scanning Symbols in a Package

5.7 The following forms are used for scanning symbols in a package. For those forms that allow you to supply a *result-form*, note that the *result-form* argument is a single form; an implicit **progn** is not generated. When the *result-form* is evaluated, the binding of *var* is set to **nil**. If the *result-form* is not supplied, the returned value is **nil**. You can use the **return** function to exit a symbol-scanning macro.

If the *body-forms* affect the accessibility of symbols in *package* (other than the one currently bound to *var*), then the effects are unpredictable.

Also note that for each of the following forms, the value of *package* defaults to the current package.

do-symbols (*var* [*package* [*result-form*]]) {*declaration*}* {*body-form*}* *[c]* Macro

This macro executes the *body-form* once for each symbol that can be found in *package* either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol, although the symbols are not bound in any particular order. Finally, the *result-form* is executed, and its values are returned. Since symbols can be present in more than one package, the *body-forms* can be executed more than once for a given symbol.

do-local-symbols (*var* [*package* [*result-form*]]) {*declarations*}* {*body-form*}* Macro

This macro executes the *body-form* once for each symbol directly present in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally, the *result-form* is executed, and its values are returned.

do-external-symbols (*var* [*package* [*result-form*]]) {*declarations*}* {*body-form*}* *[c]* Macro

This macro executes the *body-form* once for each external symbol directly present in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally, the *result-form* is executed, and its values are returned.

do-all-symbols (*var* [*result-form*]) {*declarations*}* {*body-form*}* *[c]* Macro

This macro executes the *body-form* once for each symbol present in any package. On each iteration, the variable *var* is bound to the next such symbol. Finally, the *result-form* is executed, and its values are returned.

Because a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once.

find-all-symbols *string-or-symbol* *[c]* Function

This function searches all packages in the system and returns a list of all the symbols whose print names are *string-or-symbol*. Character case is significant for *string-or-symbol*. If the value supplied for this argument is a symbol, the symbol's print name provides the string used for the search.

mapatoms *function* &optional *package inherited-p* Function

This function applies *function* to all of the symbols in *package*. The value of *function* should be a function of one argument. If *inherited-p* is non-**nil**, then the function is applied to all symbols accessible in *package*, including inherited symbols.

mapatoms-all *function* &optional *package* Function

This function applies *function* to all of the symbols in *package* and all other packages that use *package*. The *function* argument should be a function of one argument. For example:

```
(mapatoms-all
  #'(lambda (x)
       (when (alphalessp 'z x)
             (print x))))
```

Miscellaneous Package Support Functions

5.8 The following are miscellaneous package support functions.

package-name *package* *[c]* Function

This function returns the name of *package* (as a string).

package-nicknames *package* *[c]* Function

This function returns the list of nicknames (as strings) of *package*.

package-prefix-print-name *package* Function

This function returns the name to be used for printing package prefixes that refer to *package*.

rename-package *package new-name* &optional *new-nicknames* *[c]* Function

This function makes *new-name* the name for *package* and makes *new-nicknames* (a list of strings or symbols, possibly **nil**) its nicknames. An error is signaled if the new name or any of the new nicknames is already in use for another package.

find-package *name* *[c]* Function
find-package *name-or-pkg* Function

This function returns the package object whose name or one of whose nicknames is *name-or-pkg*. If no such package exists, **find-package** returns **nil**. In Common Lisp, the argument to **find-package** can be a string or a symbol. If the argument is a string, it must match the name of an existing package in a case-sensitive manner. If the argument is a symbol, the print name of the symbol is used. On the Explorer system, *name-or-pkg* can also be a package object.

pkg-find-package *name* &optional *create-p use-local-names-package* Function

This function finds or possibly creates a package named *name*. If a package whose name matches *name* is found, that package object is returned. The **find-package** function is used in the matching process. If no such package is found, a package may be created, depending on the value of *create-p* and possibly on how the user responds. The *create-p* argument must be one of the following values:

- **nil** — An error is signaled if an existing package is not found.
- **t** — A package is created and returned.
- **:find** — If the package is not found, **nil** is returned.
- **:ask** — The user is asked whether to create a package. If the answer is Yes, a package is created and returned. If the answer is No, **nil** is returned.

If a package is created, it is done by calling **make-package** with *name* as the only argument.

list-all-packages *[c]* Function

This function returns a list of all existing packages.

do-all-packages (*var* [*result-form*]) {*declaration*}* {*body-form*}* Macro

This macro executes *body-form* once for each package present in the system. On each iteration, the variable *var* is bound to the next package. Finally, the *result-form* is executed, and its value is returned.

lisp-package	Variable
ticl-package	Variable
zlc-package	Variable
system-package	Variable
keyword-package	Variable
user-package	Variable
sys:pkg-lisp-package	Variable
sys:pkg-system-package	Variable
sys:pkg-keyword-package	Variable

The values of these variables are the packages LISP, TICL, ZLC, SYSTEM, KEYWORD, and USER.

describe-package *package* Function

This function prints all available information about *package*, except for all the symbols interned in it. The *package* argument can be a package or the name of one.

In order to view all symbols interned in a package, use the following form:

(mapatoms #'print *package***)**

packagep *object* *[c]* Function

This function returns true if *object* is a package.

Final Notes on Packages

5.9 The following information provides practical notes on how your application should utilize the package system.

Common Lisp Portability Notes

5.9.1 The compiler always attempts to generate code that matches the effect of the source code. However, dealing with packages creates special problems. Because of this, Common Lisp compilers have a special dispensation. At the very least, every Common Lisp implementation guarantees that the proper object will be generated if the following forms appear only at the top level:

make-package **in-package** **shadow** **shadowing-import**
export **unexport** **use-package** **unuse-package**
import

Initialization of the Application Namespace

5.9.2 When you define an application, it is important to set up the environment correctly. To avoid environment problems, you should include the following forms, in this order, at the front of the application file:

1. Call to **provide**
2. Call to **in-package**
3. Call to **shadow**
4. Call to **export**
5. Calls to **require**
6. Calls to **use-package**
7. Calls to **import**
8. Application definitions

The **provide** and **require** functions (described in Section 23, Maintaining Large Systems) are Common Lisp functions that control the loading of files. (Also see Section 23 for a discussion of modules.) As a matter of style, each source file should contain symbols for only one package. For large applications spread over many files, a separate file should contain the package definition and declarations for all of the shadowed and external symbols. Loading this file first helps establish critical portions of the namespace for the benefit of anyone wanting to use this package.

LISTS AND LIST STRUCTURE

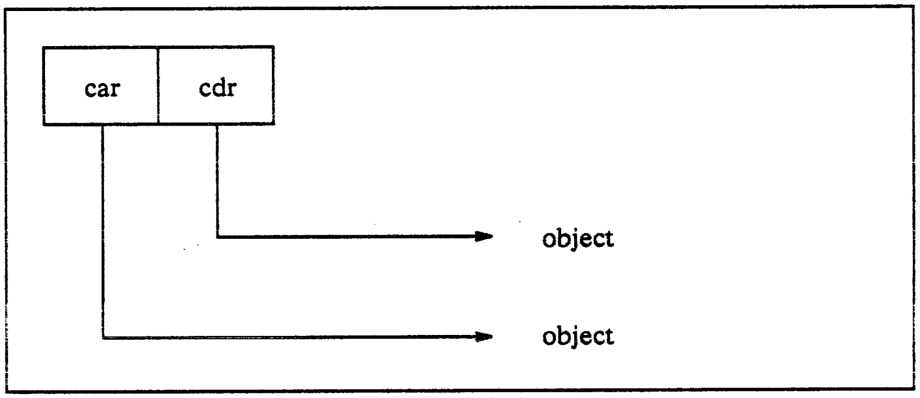
6

List Definitions

6.1 The list data type is defined to be the union of the two data types **cons** and **null**. The type **cons** is made up of data structures (also called *conses*) that have two components: a *car* and a *cdr* (see Figure 6-1). The data type **null** has only one object: **nil**, the empty list. Because **nil** is a list with no elements, it is also equivalent to the notation ( ).

Figure 6-1

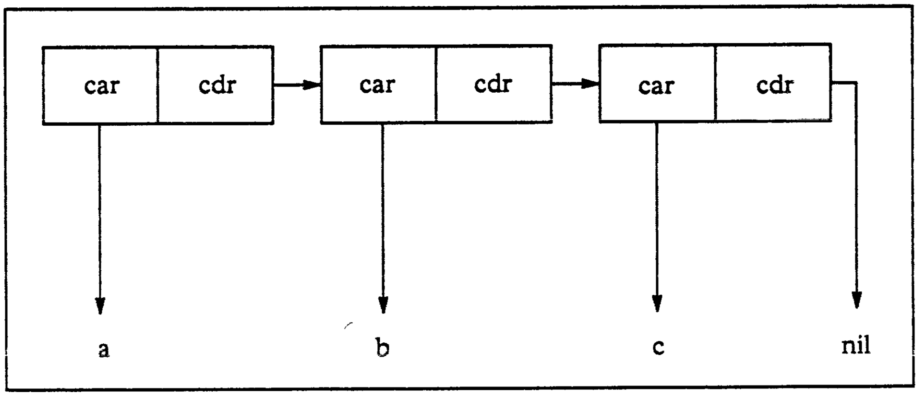
Example of a Cons



In practice, a list defines a sequence of Lisp objects. A list sequence can be nil or it can be a cons whose car is the first *element* in the sequence and whose cdr contains the rest of the sequence. The car of the last cons cell contains the last element in the list. Figure 6-2 shows the structure of the list `(a b c)`.

Figure 6-2

Example of the List (a b c)

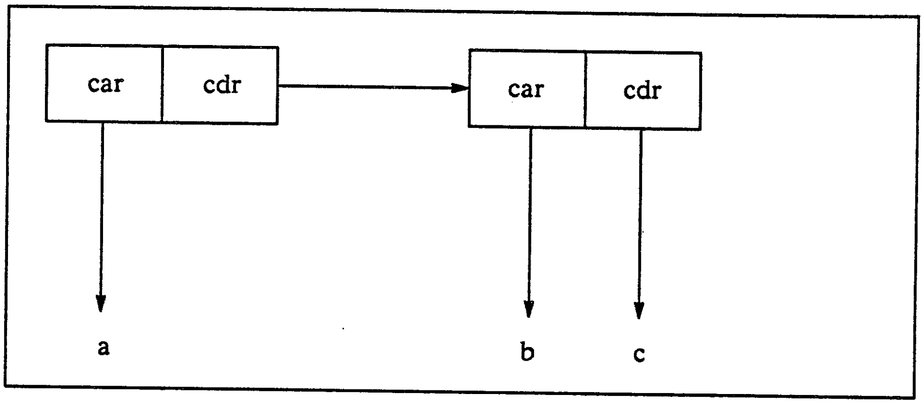


A *true list* is a list whose last cons has a cdr of **nil**. Also, by definition, **nil** is considered to be a true list.

A *dotted list* is a list in which the cdr of the last cons cell is not **nil** but is instead linked to some other object. This is called a dotted list because it is written with a dot (that is, a period) between the last two elements. You can create a dotted list by simply placing a period, surrounded by spaces, between the last two elements in a list. Figure 6-3 shows the dotted list `(a b . c)`.

Figure 6-3

Example of the Dotted List (a b . c)



Note that a single cons cell whose cdr is non-**nil** does not fit the preceding description of a list sequence. However, since such an object is of type **cons**, it is still of type **list**. Consider the following examples:

```
(a b)        ; A true list containing two elements
             ; constructed of two cons cells.

(a . b)      ; A dotted list containing one element
             ; consisting of one cons cell whose car
             ; points to a and whose cdr points to b.

(a . (b))    ; This is equivalent to (a b).
```

The last example is syntactically legal but is not a real dotted list because the cdr of the first cons is linked to another cons, and the cdr of the second (and last) cons contains **nil**. Thus, this example is actually a true list. The true list notation is used by all standard Lisp output routines whenever possible.

A *tree* is also a list, but this term is meant to include *branches* (other lists pointed to by elements of the original list), sub-branches, and so on. More precisely, a tree is composed of a cons and all other conses to which it is linked directly or indirectly via a car or cdr. Those items in the tree that are not conses are called the *leaves*. The following list is a simple example of a tree:

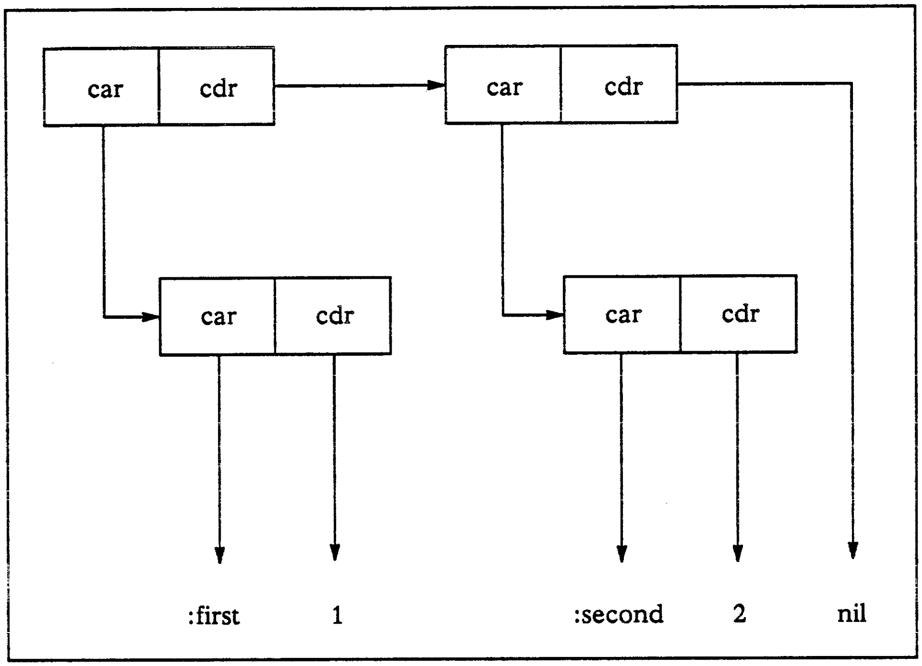
```
(a (b c) d)
```

This example is a true list with three elements, but it is also a tree with four leaves. Lisp does not have restrictions on trees in regard to regular or balanced branching. Lisp trees can even branch onto themselves; that is, the cdr of one of the cons cells can point to another cons cell that preceded it in the tree.

An *association list* (also called an *a-list* or *alist*) is a specially organized list that provides keyed access to data. An association list is a true list whose elements are cons cells. The car of each element is called the *key* and the cdr is called the *datum*. Both the key and the datum can be any kind of Lisp object; however, the key is treated as an identifying label for the associated datum. Figure 6-4 shows the following association list:

```
((:first . 1) (:second . 2))
```

Figure 6-4 Example of the Association List ((:first . 1) (:second . 2))



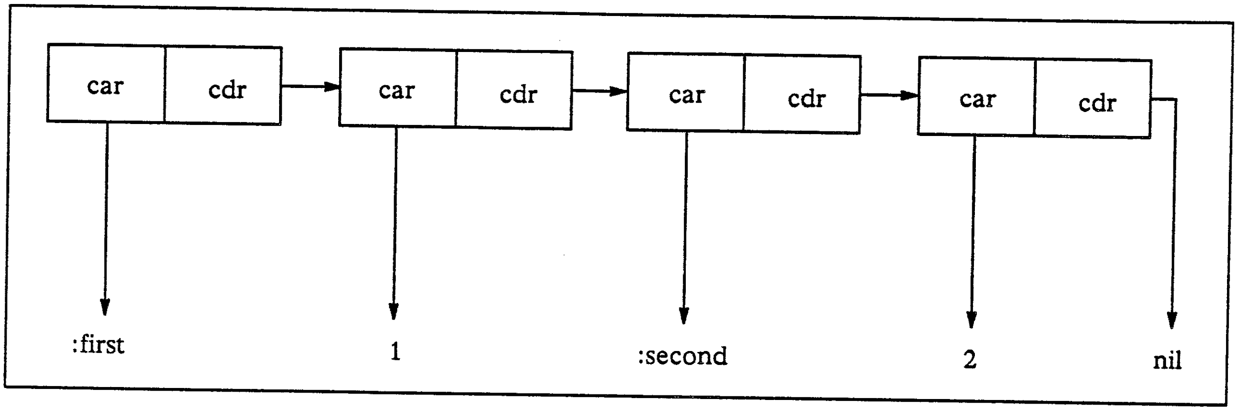
Several elements in an association list can have the same key; but the searching functions always search from the front of the association list, so only the first element with that key is found. For example, in the following association list, a function looking for `:key1` would always find `(:key1 . x)` rather than `(:key1 . y)`:

```
((:key1 . x) (:key1 . y) (:key2 . z))
```

A *property list* (also called a *plist*) is another specially organized list that provides keyed access to data. While property lists are similar in principle to a-lists, the form of property lists and the support functions that operate on them are different. Property lists are true lists whose elements are treated as paired items: the first item is the identifying key called an *indicator*, and the second item is a Lisp object called the *value* or *property*. Thus, there must be an even number (including 0) of elements in a property list. Each indicator should occur only once in the list. Figure 6-5 shows the following property list:

```
(:first 1 :second 2)
```

Figure 6-5 Example of the Property List (:first 1 :second 2)



One of the first uses of property lists was to keep track of associated information for symbols. Through the use of property lists, each symbol would own a unique set of tabular data. Because of this implied uniqueness of data, most functions that change property lists do so in a destructive manner because it was originally assumed that no other references to this list would exist. See the discussion of altering data structures in paragraph 6.6, Altering List Structure.

A *circular list* is a list in which the cdr of one of the cons cells points to another cons cell that appears earlier in the list.

Note that true lists, dotted lists, trees, a-lists, plists, and circular lists are merely descriptive terms and do not constitute data types themselves. Certain functions, however, do specify that their arguments must be lists with these respective attributes. Whenever the term *list* is used without qualification, it is assumed to mean true lists.

Cdr-Coding

6.2 *Cdr-coding* is an internal storage technique used to store conses in the Explorer system. You need not be concerned about cdr-coding unless you require extra storage efficiency in your program.

Why is cdr-coding important to users? In fact, it is all transparent to you; everything works the same way whether or not compact representation is used, from the perspective of the semantics of the language. That is, the only difference that cdr-coding makes is a difference of efficiency. The compact representation is more efficient in most cases. However, if the conses are to have **rplacd** executed on them, then invisible pointers will be created, extra memory will be allocated, and the compact representation will degrade storage efficiency rather than improve it. Also, accesses that go through invisible pointers are somewhat slower because more memory references are needed. Thus, if storage efficiency is of great concern, you should be careful about which lists are stored in which representations.

The usual and obvious internal representation of a cons in any implementation of Lisp is as a pair of pointers, adjacent in memory. If the amount of storage required to store a Lisp pointer is called a *word*, then conses normally occupy two words. One word (conceptually, the first) holds the car, and the other word holds the cdr. To access the car or cdr of a list, you simply refer to this memory location; to change the car or cdr, you simply store into this memory location.

Very often, conses are used to store lists. If the above representation is used, a list of *n* elements requires two times *n* words of memory: *n* to hold the pointers to the elements of the list, and *n* to point to the next cons or **nil**. To optimize this particular case of using conses, the Explorer system uses a storage representation called cdr-coding to store lists. The basic goal is to allow a list of *n* elements to be stored in only *n* locations, while allowing conses that are not parts of lists to be stored in the usual way.

Every word of memory has an extra two-bit field called the *cdr-code* field. This field can have one of three values: **cdr-normal**, **cdr-next**, or **cdr-nil**. In the normal method of storing a cons described previously, the cdr-code of the first word is **cdr-normal**. (The cdr-code of the second word is insignificant; it is never looked at.) The cons is represented by a pointer to the first of the two words. When a list of *n* elements is stored in the most compact way, pointers to the *n* elements occupy *n* contiguous memory locations. The cdr-codes of all these locations are **cdr-next**, except for the last location whose cdr-code is **cdr-nil**. The list is represented as a pointer to the first of the *n* words.

Given this data structure, finding the car for a particular list is easy: you simply read the contents of the location addressed by the pointer. Finding the cdr is more complex. First, you must read the contents of the location addressed by the pointer and inspect the cdr-code found there. If the code is **cdr-normal**, then you add 1 to the pointer, read the location it addresses, and return the contents of that location; that is, you read the second of the two words. If the code is **cdr-next**, you add 1 to the pointer and simply return that pointer without doing any more reading; that is, you return a pointer to the next word in the *n*-word block. If the code is **cdr-nil**, you simply return **nil**.

These rules work fine even if you mix the two kinds of storage representation within the same list.

What about changing the structure? Like **car**, **rplaca** is very easy: you simply store into the location addressed by the pointer. To use **rplacd**, you must read the location addressed by the pointer and examine the cdr-code. If the code is **cdr-normal**, you simply store into the location 1 greater than that addressed by the pointer; that is, you store into the second of the two words. But if the code is **cdr-next** or **cdr-nil**, the memory cell normally reserved for storing the cdr of the cons does not exist because this is the cell that has been optimized out.

However, this problem can be solved by the use of *invisible pointers*. An invisible pointer is a special kind of pointer recognized by its data type (Explorer pointers include a data type field as well as an address field). When the Explorer system reads a word from memory and this word is an invisible pointer, the system reads the word pointed to by the invisible pointer and uses this word instead of the invisible pointer itself. Similarly, when the system writes to a location, it first reads the location. If this location contains an invisible pointer, the system writes to the location addressed by the invisible pointer instead. (This is a somewhat simplified explanation; actually, there are several kinds of invisible pointers that are interpreted in different ways at different times and used for tasks other than cdr-coding.)

The following explanation describes what **rplacd** does when the cdr-code is **cdr-next** or **cdr-nil**; in this explanation, call the location addressed by the first argument to **rplacd** *lst*. First, you allocate two contiguous words in the same area that *lst* points to. Then you store the old contents of *lst* (the car of the cons) and the second argument to **rplacd** (the new cdr of the cons) into these two words. You set the cdr-code of the first of the two words to **cdr-normal**. Then you write an invisible pointer, pointing at the first of the two words, into location *lst*. (It does not matter what the cdr-code of this word is because the invisible pointer data type is checked first.)

Whenever any operation is performed on the cons (via **car**, **cdr**, **rplaca**, or **rplacd**), the initial reading of the word pointed to by the Lisp pointer that represents the cons finds an invisible pointer in the addressed cell. When the invisible pointer is seen, the address it contains is used in place of the original object.

You should try to use the normal representation for those data structures that will be subject to **rplacd** operations, including **nconc** and **nreverse**, and use the compact representation for other structures. The functions **cons**, **ncons**, and their area variants make conses in the normal representation. The functions **list**, **list***, **list-in-area**, **make-list**, and **append** use the compact representation. The other list-creating functions, including **read**, currently make normal lists, although this may be changed. Some functions, such as **sort**, take special care to operate efficiently on compact lists (**sort** effectively treats them as arrays). The **nreverse** function is rather slow on compact lists, currently, because it simple-mindedly uses **rplacd**, but this may be changed.

Copying a list and converting it into compact form can be achieved with **copy-list**.

Functions Associated With Conses

6.3 The following functions are used for manipulating conses. In these function descriptions, the term *cons* can refer to any kind of cons cell, which includes non-**nil** lists.

car *list*

[c] Function

This function returns the object in the car cell of *list*. The *list* argument must be a cons or **nil**. Note that the car of **nil** is defined to be **nil**. For example:

```
(car '(x y z)) => x
```

cdr *list*

[c] Function

This function returns the object pointed to by the cdr cell of *list*. If *list* is a list, this cdr cell may point to another cons cell. The argument *list* must be a cons or **nil**. The cdr of **nil** is defined to be **nil**. For example:

```
(cdr '(x y z)) => (y z)
(cdr '(x . y)) => y
```

caar *list* *[c]* Function
cadr *list* *[c]* Function
cdar *list* *[c]* Function
cddr *list* *[c]* Function
caaar *list* *[c]* Function
caadr *list* *[c]* Function
cadar *list* *[c]* Function
caddr *list* *[c]* Function
cdaar *list* *[c]* Function
cdadr *list* *[c]* Function
cddar *list* *[c]* Function
cdddr *list* *[c]* Function
caaaar *list* *[c]* Function
caaadr *list* *[c]* Function
caadar *list* *[c]* Function
caaddr *list* *[c]* Function
cadaar *list* *[c]* Function
cadadr *list* *[c]* Function
caddar *list* *[c]* Function
cadddr *list* *[c]* Function
cdaaar *list* *[c]* Function
cdaadr *list* *[c]* Function
cdadar *list* *[c]* Function
cdaddr *list* *[c]* Function
cddaar *list* *[c]* Function
cddadr *list* *[c]* Function
cdddar *list* *[c]* Function
cddddr *list* *[c]* Function

These functions represent combinations of cars and cdrs. The names of these functions begin with **c** and end with **r**, and in between is a sequence of **a** and **d** letters corresponding to a single car or cdr operation. For instance, note the following equivalent forms:

```
(cddadr x) <=> (cdr (cdr (car (cdr x))))
```

For any list, you can use **cadr** to return the second element at the top level, **caddr** to return the third, and **cadddr** to return the fourth. If, for example, the second element of a list is itself a list, then you can use **caadr** to return the first element of the sublist, **cadadr** to return the second element of the sublist, and so on (see the corresponding functions **first** and **second**).

All of these functions are acceptable to **setf** as *place* forms.

nthcdr *count list* *[c]* Function

This function returns the *n*th cdr of the list, where *count* is an integer greater than or equal to 0. If *count* is 0, **nthcdr** returns the entire list; if *count* is greater than the length of *list*, **nil** is returned.

This function is acceptable to **setf** as a *place* form.

car-safe *object* Function
cdr-safe *object* Function
caar-safe *object* Function
cadr-safe *object* Function
cdar-safe *object* Function
cddr-safe *object* Function
nth-safe *count object* Function
nthcdr-safe *count object* Function

These functions return the same values as the corresponding non-safe functions (that is, **car**, **cdr**, and so on), except that they return **nil** where the corresponding non-safe function would produce an error. The **nth-safe** and **nthcdr-safe** functions include a *count* argument to specify which car or cdr is to be returned. These functions are about as fast as the non-safe functions. You could get the same result, though more slowly, by handling the **sys:wrong-type-argument-error** condition. Consider the following examples:

```
(car-safe '(a . b)) => a
(car-safe nil) => nil
(car-safe 'a) => nil
(car-safe "yabba") => nil
(cadr-safe '(a . b)) => nil
(cadr 3) => ERROR
(cadr-safe 3) => nil
```

cons *car-val cdr-val* *[c]* Function

This function is a primitive function that returns a cons of its two arguments, with *car-val* in the car cell and *cdr-val* in the cdr cell. The arguments can be any Lisp object. For example:

```
(cons 'a 'b) => (a . b)
(cons 'a '(b c)) => (a b c)
```

The second line of code shows that **cons** can be thought of as a function that produces a list with a new element on the front.

cons-in-area *car-val cdr-val area* Function

This function constructs a cons in a specific *area*. (Areas are an advanced feature of storage management, as explained in Section 25, Storage Management; if you are not concerned with functions that deal with areas, then you can disregard them.) The first two arguments can be any Lisp object, but the third must be the number of an area in which to construct the cons.

Functions Associated With Lists

6.4 The following functions are used for manipulating lists. In these function descriptions, when a function is said to be *destructive*, it means that the function permanently changes the values of its arguments. A *nondestructive* function merely makes copies of the values of its arguments and uses these copies during its execution. When a nondestructive function returns, the original values of its arguments are unchanged. However, this does not mean that a copy of the original values was made.

list-length *list* *[c]* Function

This function returns either the length of the list or **nil** if the list is circular. This function is used chiefly to determine if a list is circular. For example:

```
(list-length ´( )) => 0
(list-length ´(ace ace jack 10 5)) => 5
(list-length ´(tarzan (jane cheetah) boy)) => 3

(setq g-pig (list ´x ´y ´z))   ; Let g-pig be the list (x y z).

(setf (cdddr g-pig) g-pig)     ; Let g-pig become a circular list.

(list-length g-pig) => nil
```

See also **length** in paragraph 9.3, Elementary Sequence Functions.

nth *count list* *[c]* Function

This function returns the *n*th (zero-based) element of *list* or **nil** if *count* is greater than or equal to the length of *list*. This function can be used as the *place* argument to **setf**. Consider the following examples:

```
(nth 0 ´(a b c)) => a
(nth 2 ´(a b c)) => c
(nth 3 ´(a b c)) => nil
```

Note that the order of the arguments for **nth** is the opposite of that for **elt**: for **nth**, you specify the element index before the list in which the element is to be found.

first *list* *[c]* Function
second *list* *[c]* Function
third *list* *[c]* Function
fourth *list* *[c]* Function
fifth *list* *[c]* Function
sixth *list* *[c]* Function
seventh *list* *[c]* Function
eighth *list* *[c]* Function
ninth *list* *[c]* Function
tenth *list* *[c]* Function

These functions take a list as an argument and return the appropriate element of the list. The function **first** is identical to **car**; **second** is identical to **cadr**, and so on. These names are provided because they make more sense when you are thinking of the argument as a list rather than just as a cons. Note that the numbering of list elements starts with 1 as opposed to starting with 0 as the numbering of the function **nth** does. Note the following equivalences:

```
(fourth x) <=> (nth 3 x) <=> (cadddr x)
```

The **setf** macro can be used with each of these functions as *place* forms to store a value into the indicated position of a list.

rest *list* *[c]* Function

This function does the same thing as **cdr**. It returns the rest of *list* after the first element. It can be used as the *place* argument of **setf**.

last *list* *[c]* Function

This function returns the last cons of *list*. This function returns **nil** if *list* is **nil**. For example:

```
(setq lst '(x y z))
(last lst) => (z)
(rplacd (last lst) '(a b))
lst => (x y z a b)
(last '(x y . z)) => (y . z)
```

A common way to return the last element from a list is with the following form:

```
(car (last some-list))
```

list &rest *objects* *[c]* Function

This function constructs a list by using its arguments as the elements of the list. For example:

```
(list 'x 'y 'z '(1 2) 3) => (x y z (1 2) 3)
(list 'x) => (x)
(list) => nil
```

list* *object* **&rest** *others* *[c]* Function

This function constructs a list whose last two elements form a dotted pair, unless the last element is itself a list. This function must be given at least one argument, but if **list*** is given only one argument, it returns that argument (unlike **list**, which returns a single argument as a list). For example:

```
(list* 'w 'x 'y 'z) => (w x y . z)

; The preceding produces the same result as the following expression.
(cons 'w (cons 'x (cons 'y 'z)))

; Note also the following special cases:
(list* 'u 'v 'w '(x y z)) => (u v w x y z)
(list* 'z) <=> z
```

make-list *size* **&key :initial-element** *[c]* Function
make-list *size* **&key :initial-element :area** Function

This function makes and returns a list containing the number of elements specified by *size*, each of which is initialized to the value of **:initial-element**. If no **:initial-element** is provided, each element in the list is initialized to **nil**. The *size* argument must be a nonnegative integer.

The keyword **:area** is an Explorer extension to Common Lisp, and if this keyword is specified, this function makes the list in the specified area. Consider the following examples:

```
(make-list 6) => (nil nil nil nil nil nil)
(make-list 4 :initial-element 'xyz) => (xyz xyz xyz xyz)
```

list-in-area *area* &rest *objects* Function

This function is the same as **list**, except that it constructs a list in a specified area. (Areas are discussed in paragraph 25.5, Storage and Allocation Areas.)

list*-in-area *area* &rest *objects* Function

This function is the same as **list***, except that the list it constructs is made in a specified area.

circular-list &rest *objects* Function

This function constructs a circular list with the elements specified by *objects*. This function is the same as **list**, except that the constructed list is used as the last cdr instead of **nil**. This function is particularly useful with **mapcar**, as shown in the following example.

The following form returns a list whose elements are the sum of the corresponding elements in each of the argument lists:

```
(setf foo '(1 2 3))
(mapcar #'+ foo '(5 5 5))
```

Now consider the following use of circular lists:

```
(mapcar #'+ foo (circular-list 5))
```

This form adds each element of `foo` to `5` without needing to know the length of `foo`.

copy-list *list* *[c]* Function

This function creates and returns a list that is **equal** (not **eq**) to *list*. Note that individual elements of the new list *are* **eq** to the corresponding elements of *list*. The *list* argument must be a cons or a list.

This function copies only the top level of *list* (for copying all levels of a list structure, see **copy-tree** later in this section). If *list* is a dotted list, then the copy is also a dotted list. The **copy-list** function actually creates a copy of the argument list that has gone through a process of memory compaction. Therefore, the copied list takes up less memory. This compaction process is called cdr-coding, and several functions use this process for increased memory efficiency (see paragraph 6.2, Cdr-Coding).

copylist* *list* Function

This function is the same as **copy-list**, except that the last cons of the resulting list is never cdr-coded. This feature is advantageous if the list is later used as the first argument to **nconc**, because altering the cdr of a cdr-coded cons is less efficient.

copy-tree *tree* *[c]* Function

This function copies complete tree structures. Any Lisp object can be used for the *tree* argument. This function returns an **equal** copy of *tree* unless this argument is not a cons, in which case **copy-tree** simply returns *tree* itself. The **copy-tree** function operates by recursively copying all the conses in *tree*, halting the recursion whenever it finds something that is not a cons. Note that all leaves of the new tree are **eq** to the corresponding leaves of *tree*. The function **copy-tree** does not preserve list circularity and substructure sharing.

This function uses cdr-coding for maximum memory efficiency.

append &rest *objects* *[c]* Function

This function returns a list that is the concatenation of its arguments. All of the arguments to **append**, except the last one, must be lists; the original lists are not destroyed. For example:

```
(append '(a b c) '(d e f) nil '(g)) => (a b c d e f g)
```

The **append** function makes copies of the conses of all the lists it is given, except for the last one. Therefore, the new list shares the conses of the last argument to append, but all of the other conses are newly created. Only the lists are copied, not the elements of the lists.

The last element in *objects* can be any Lisp object—it need not be a list.

The last argument becomes the end of the new list. For example:

```
(append '(a b c d) 'e) => (a b c d . e)
```

The definition of **append** minimizes storage utilization by turning all the arguments that are copied into one cdr-coded list.

revappend *list1 list2* Function

This function first constructs a list where the element sequence of *list1* is reversed and then concatenates this list with *list2*. Both *list1* and *list2* should be lists, and they are copied, not destroyed. For example:

```
(setq list1 '(a b c))
(setq list2 '(d e f))
(revappend list1 list2) => (c b a d e f)
```

Compare this function with **nreconc**.

push *item place* *[c]* Macro

This macro stores *item* onto the front of the list stored at *place*. The argument *place* must be a generalized variable acceptable to **setf** and must contain a list. In other words, one new cons cell is allocated whose car is *item* and whose cdr is the list stored at *place*. The new list is then stored back into *place* and returned. If the list stored at *place* is considered a stack, then **push** pushes *item* onto the stack. For example:

```
(setq a '(1 2 (b c) d))

(push 3 (third a)) => (3 b c)

;; Now a is changed.
a => (1 2 (3 b c) d)
```

pushnew *item place* &key :test :test-not :key *[c]* Macro

This macro pushes *item* onto the list stored at *place* and returns the new list. If *item* already belongs to the list, then the list is returned unchanged. The argument *item* can refer to any Lisp object.

The :test .and :test-not keywords are used to specify comparison tests. The :key keyword is used to specify a function that preprocesses the items in the list before the comparison is made. The style of these arguments is similar to other generic sequence functions. The functionality of the **:key** argument is slightly different for the **pushnew** function because this function is applied to the *item* argument as well as each element in *place*. The form *place* must be a generalized variable acceptable to **setf** and must contain a list.

If the list is considered a set, **pushnew** acts like the function **adjoin**. Consider the following example:

```
;; Set value of x to a list.
(setf x '(a b (c d) e))

;; Push number 7 onto the front of sublist of x
;; if it is not there.
(pushnew 7 (third x)) => (7 c d)

;; x is now changed.
x => (a b (7 c d) e)

;; Push c to front of sublist if it is not there.
(pushnew 'c (third x)) => (7 c d)

;; c already in sublist of x, so x remains unchanged.
x => (a b (7 c d) e)
```

pop *place* *[c]* Macro

This macro returns the car of *place*. The cdr of the original value for *place* becomes the new value of *place*. The value specified for *place* must be a cons. For example:

```
(setf top '(cola root-beer ginger-ale))
(pop top) => cola
top => (root-beer ginger-ale)
```

butlast *list* &optional *count* *[c]* Function

This function returns a copy of *list* with the last *count* elements deleted. This function is nondestructive; thus, *list* itself is not changed. The default for *count* is 1. Because the elements of *list* are indexed starting with 0, if *count* is specified and *list* has *count* or fewer elements, **butlast** returns **nil**. For example:

```
(setf thud '(bonk crash ping pong))
(butlast thud) => (bonk crash ping)
thud => (bonk crash ping pong)

;; Last is (3 4); return copy of all but (3 4).
(butlast '((1 2) (3 4))) => ((1 2))

;; count is equal to the number of elements in the list.
;; Thus, butlast returns nil.
(butlast '(not-least) 1) => nil

;; count is greater than the number of elements in the list.
;; Thus, butlast returns nil.
(butlast nil 1) => nil
```

firstn *count list* Function

This function returns a list of length *count* whose elements are the first *count* elements of *list*. If *list* is fewer than *count* elements long, then the remaining elements of the returned list are **nil**. For example:

```
;; Return first two elements.
(firstn 2 '(a b c d)) => (a b)

;; Return no elements, that is, nil.
(firstn 0 '(a b c d)) => nil

;; Return first 6 elements.
(firstn 6 '(a b c d)) => (a b c d nil nil)
```

nleft *count list* &optional *tail* Function

This function returns a list containing the last *count* elements of *list*. If *count* is too large, the function returns *list*.

The form (**nleft** *count list tail*) returns a tail of *list* such that taking *count* more cdrs would yield *tail*. When *tail* is **nil**, the **nleft** function is the same as the two-argument case. If *tail* is not **eq** to any tail of *list*, **nleft** returns **nil**. Consider the following example:

```
(setf x '(1 2 3 4 5)
     y (cdddr x)) => (4 5)

(nleft 2 x y) => (2 3 4 5)
```

ldiff *list tail* *[c]* Function

This function (meaning *list difference*) returns a list containing all the elements of *list* positioned before *tail*. This function is not destructive. The test **eq** is used to compare *list* and *tail*. The argument *list* must be a list, and *tail* should be one of the conses that make up *list*. If *tail* is not a cons of *list* or if *tail* is nil, then **ldiff** returns a copy of *list*. In any case, the list specified by *list* is unchanged. For example:

```
(setf notes '(a b c d e)) => (a b c d e)
(setf high-notes '(cddr notes) => (c d e)

;; Find the difference between notes and high-notes.
(ldiff notes high-notes) => (a b)

;; Now set new-high-notes to a different set (not eq) of cons cells
;; pointing to (c d e) and try ldiff again.
(setf new-high-notes '(c d e)) => (c d e)
(ldiff notes new-high-notes) => (a b c d e)
```

Stack Lists

6.5 When you are creating a list that will not be needed once the function that creates it is finished, you can create the list on the stack instead by consing it. This method avoids any permanent storage allocation because the space is reclaimed as part of exiting the function. However, this method is also risky: if any pointers to the list remain after the function exists, they become meaningless.

These lists are called *temporary lists* or *stack lists*. You can create them explicitly using the special forms **with-stack-list** and **with-stack-list***. Some &rest arguments also create stack lists.

If a stack list or a list that might be a stack list is to be returned or made part of permanent list structure, it must first be copied (see **copy-list** in paragraph 6.4, Functions Associated With Lists). The system cannot detect the error of omitting to copy a stack list; you may simply find that a value has changed without you knowing it.

with-stack-list (*variable* {*element*}*) {*body-form*}* Special Form
with-stack-list* (*variable* {*element*}* *tail*) {*body-form*}* Special Form

These special forms create stack lists that reside inside the stack frame of the function in which these special forms are used. You should assume that the stack lists are valid only until the special form is exited. For example:

```
(with-stack-list (funky x y)
   (get-down funky))
```

The following form is equivalent to the preceding except that `funky`'s value in the first example is a stack list:

```
(let ((funky (list x y)))
   (get-down funky))
```

The list created by **with-stack-list*** looks the one created by **list***. The value of *tail* becomes the ultimate cdr rather than an element of the list.

The following is a practical example. The **condition-resume** macro (see Section 21, Compiler Operations) could have been defined as follows:

```
(defmacro condition-resume (handler &body body)
   `(with-stack-list* (eh:condition-resume-handlers
                          ,handler eh:condition-resume-handlers)
       . ,body))
```

It is an error to execute **replacd** on a stack list (except for the tail of one made using **with-stack-list***). However, **rplaca** works normally on a stack list.

Altering List Structure

6.6 The following set of functions alter a list's structure in a destructive manner. For more information on destructive and nondestructive functions, see paragraph 6.4, Functions Associated With Lists.

Altering a list's structure can be done safely, but be especially careful when doing so. Specifically, suppose that several variables point to the same list object. If you use one of these variables in a function that modifies the list destructively, then all the variables that pointed to the original list now point to the altered list. In other words, a destructive function changes the list's cons cells permanently. Under certain circumstances this alteration may be desirable, but in some cases destructive functions may lead to quite unexpected results. However, using destructive functions has two advantages: destructive functions are usually faster because they change a list instead of making a copy and modifying this copy, and destructive functions perform less consing, which means that less garbage is generated in your virtual address space.

Also note that you should not use these functions for side effects. If you are trying to change a variable's value, you should explicitly set the new value. For instance, suppose you want to destructively modify a list `x` to find the intersection of `x` and `y`. Assume that `x` is set to the list `(1 2 3)` and that `y` is set to the list `(2 3)`. The following, then, does not change `x`:

```
(nintersection x y) => (2 3)
x => (1 2 3)
```

The correct way to modify `x` is as follows:

```
(setf x (nintersection x y)) => (2 3)
```

nbutlast *list* &optional *count* *[c]* Function

This function is the destructive version of **butlast** (see paragraph 6.4, Functions Associated With Lists). The **nbutlast** function returns *list* with the last *count* elements deleted. The default for *count* is 1. Because the elements of *list* are indexed starting with 0, if *count* is specified and *list* has *count* or fewer elements, **nbutlast** returns **nil** and *list* is unchanged. Use this function only for its returned value, not for its side effects. Consider the following example:

```
(setf thud '(bonk crash ping pong whizz))
(setf thud (nbutlast thud)) => (bonk crash ping pong)
thud => (bonk crash ping pong)
(setf thud (nbutlast thud 2)) => (bonk crash)

(setf thud '(bonk crash . ping))
(nbutlast thud) => (bonk)

;; count is equal to the number of elements in the list.
;; Thus, nbutlast returns nil.
(nbutlast '(not-least)) => nil

;; count is greater than the number of elements in the list.
;; Thus, nbutlast returns nil.
(nbutlast nil) => nil
```

nconc &rest *lists* *[c]* Function

This function is similar to **append** in that it concatenates all its arguments into one list but differs in that all its arguments are altered to create the new returned list. Unlike **append**, **nconc** does not make copies of its arguments. Use this function only for its returned value, not for its side effects. Consider the following example:

```
(setq a '(u v w))
(setq b '(x y z))
(setf a (nconc a b)) => (u v w x y z)
a => (u v w x y z)
```

In this example, the value of `a` changes because `nconc` causes the cdr pointer of its last cons to point to the value of `b`. Evaluating `(nconc a b)` again would produce a *circular* list structure. This result is produced because the last cons of `a` is the value of `b`, and evaluating this form again causes the cdr pointer of this last cons to point to itself. Were the ***print-circle*** global variable non-**nil**, its printed representation would be `(u v w . #1=(x y z . #1#))`.

nreconc *list1 list2* *[c]* Function

The form `(nreconc list1 list2)` is a more efficient version of `(nconc (nreverse list1) list2)`. The arguments *list1* and *list2* must be lists, and **nreconc** alters the *list1* argument. Use this function only for its returned value, not for its side effects. See also **revappend**.

rplaca *cons object* *[c]* Function

This function destructively replaces the car of *cons* with *object* and returns *cons*. For example:

```
(setf lst '(a b c))

;; Replace first element of lst with x.
(rplaca lst 'x) => (x b c)
lst => (x b c)

;; Replace second element of lst with y.
(rplaca (rest lst) 'y) => (y c)
lst => (x y c)
```

You can use the **setf** macro instead of **rplaca**. Using **setf** is probably clearer than **rplaca**. For example:

```
(setf lst '(a b c))

;; Replace first element of lst with x.
(setf (first lst) 'x)
lst => (x b c)
```

rplacd *cons object* *[c]* Function

This function alters the cdr of *cons* with *object* and returns *cons* with this modification. For example:

```
;; Set lst to (a b c).
(setf lst '(a b c))

;; Replace cdr of lst with (x y).
(rplacd lst '(x y)) => (a x y)
lst => (a x y)

;; Replace cdr of lst with z.
(rplacd lst 'z) => (a . z)
lst => (a . z)
```

You can use the **setf** macro instead of **rplacd**. Using **setf** is probably clearer than using **rplacd**. For example:

```
(setf lst '(a b c))

(setf (rest lst) '(x y))
lst => (a x y)

(setf (rest lst) 'z)
lst => (a . z)
```

List Functions With Keyword Arguments

6.7 Several functions that operate on lists can also accept one or more keyword arguments that modify the function in various ways. The basic functions fall into two categories: those that operate on every occurrence of a single target element in a list, and those that operate on every element that satisfies a specified predicate. As an example of the first category, the following form returns the sublist of `example-list`, which begins with `target-element`:

```
(member target-element example-list)
```

Many of the list functions from the second category are variants of the first category with an added **-if** or **-if-not** suffix. Rather than expecting a target element, these functions expect a test condition predicate as a required argument. For example, the following form returns the tail of `example-list` whose first element is a number:

```
(member-if #'numberp example-list)
```

The following keywords can be used to specify arguments that modify the operation performed by many of the list functions:

- **:test, :test-not** — The argument to either of these keywords is a function that determines which elements in the list are to be operated on. This function should expect two arguments. The order of the arguments supplied to the test function is the same as the order of the arguments to the calling list function. In most cases, this means that the first argument is the target item and the second argument is an item from the list. When no **:test** argument is specified, the default is **eql**. For example, the following form returns the sublist, or tail, whose first element is `ok-symbol`:

  ```
  (member 'ok-symbol
          '(good-symbol ok-symbol bad-symbol)
          :test #'eq)
  => (ok-symbol bad-symbol)
  ```

  If **:test-not** had been used in the previous example, the result would have been the original list, because the first element in the list is not `eq` to `ok-symbol`.

- **:key** — The argument to this keyword is a function that preprocesses every element in the list before the condition test is applied. The result of this function is passed as an argument to the test condition predicate. The function specified should expect only one argument. When no **:key** function is specified, the default is to use the element in the list as is. For example, the following form returns the sublist in which the `car` of the first element is `eq` to `ok-symbol`:

  ```
  (member 'ok-symbol
         '((good-symbol . good-data)
           (ok-symbol . ok-data)
           (bad-symbol . bad-data))
         :key #'car
         :test #'eq)
  => ((ok-symbol . ok-data) (bad-symbol . bad-data)
  ```

NOTE: These functions resemble several functions documented in Section 9, Sequences; however, the functions documented in this section operate only on lists.

Substitution Within a List

6.7.1 The following functions provide various ways to change list elements.

subst *new old tree* &key **:test :test-not :key** — *[c]* Function
subst-if *new test tree* &key **:key** — *[c]* Function
subst-if-not *new test tree* &key **:key** — *[c]* Function

The function **subst** replaces with *new* every element or subtree in *tree* that matches *old*, returning a new tree. This function is not destructive; therefore, list structure is copied as necessary to avoid destroying parts of *tree*. For example:

```
(setf x '(1 2 3 4 5)) => (1 2 3 4 5)
(setf y (subst 8 3 x)) => (1 2 8 4 5)
(eq (cdddr x) (cdddr y)) => true

(setf x '((1 2) (3 4))) => ((1 2) (3 4))
(setf y (subst '(4 5 6) '(3 4) x :test #'equal))
=> ((1 2) (4 5 6))
(eq (car x) (car y)) => true
```

The rule for copying portions of a tree is as follows: if a branch of a tree has no substitution performed in it, then that cons cell is **eq** to the respective branch in the original tree. Otherwise, the tree is copied. Consider the following example:

```
(subst 'peppers 'pebbles
       '(Peter Piper picked a peck of (pickled pebbles)))
=> (Peter Piper picked a peck of (pickled peppers))

(subst 'ha nil '(Peter Piper picked a peck of
                        (pickled peppers)))
=> (Peter Piper picked a peck of (pickled peppers . ha) . ha)

(subst '(something . blue) '(something . black)
  '((something . old) ((something . new) something . borrowed)
    (something . black)) :test #'equal)
=> ((something . old) ((something . new) something . borrowed)
    (something. blue))
```

Compare this function with **substitute.**

The functions **subst-if** and **subst-if-not,** which are variations of **subst,** use a test function that takes a single argument and is applied to each element of the tree. Whether the substitutions are performed depends on the results of *test*. In **subst-if,** they are performed if *test* is true, and in **subst-if-not,** they are performed if *test* is not true.

NOTE: The **subst** function has a slightly different meaning in Zetalisp mode; see Appendix A.

---

nsubst *new old tree* &key :test :test-not :key *[c]* Function
nsubst-if *new test tree* &key :key *[c]* Function
nsubst-if-not *new test tree* &key :key *[c]* Function

These functions perform the same operations as **subst** but destructively alter the value of *tree*. Thus, **nsubst** substitutes *new* for every occurrence of *old* in *tree*. Use these functions only for their returned values, not for their side effects.

sublis *a-list tree* &key :test :test-not :key *[c]* Function

This function performs multiple parallel substitutions for objects in *tree*, returning a new tree. The argument *tree* is not modified because the list structure is copied as necessary. If no substitutions are made, the result is *tree*. The argument *a-list* is an association list (a list of pairs). Each element of *a-list* specifies one replacement; the car is what to look for, and the cdr is what to replace it with. The first argument to the test function is the car of an element of *a-list*. For example:

```
(sublis '((a . 1) (b . 4) (c . 10))
         '(+ (* a (^ x 2)) (* b x) c))
=> (+ (* 1 (^ x 2)) (* 4 x) 10)
```

nsublis *a-list tree* &key :test :test-not :key *[c]* Function

This function performs the same operations as **sublis** but destructively alters *tree*. Use this function only for its returned value, not for its side effects.

Lists as Sets

6.7.2 The following functions provide a variety of operations for treating lists as sets.

member *item list* &key :test :test-not :key *[c]* Function
member-if *predicate list* &key :key *[c]* Function
member-if-not *predicate list* &key :key *[c]* Function

The **member** function determines if *item* is a member of *list* according to **:test**, which defaults to **eql**. If *item* is a member of *list*, **member** returns the rest of *list* beginning with *item*. If *item* is not a member of *list*, **member** returns **nil**; thus, this function can be used as a predicate. The keywords are described in paragraph 6.7, List Functions With Keyword Arguments.

NOTE: The **member** function has a slightly different meaning in Zetalisp mode; see Appendix A.

Consider the following example:

```
(member 'x '(a b c d)) => nil
(member-if #'numberp '(a 5/3 foo)) => (5/3 foo)
(member-if-not #'numberp '(a 5/3 foo)) => (a 5/3 foo)
(member 'a '(g (a y) c a d x a z)) => (a d x a z)
```

The result of the last example is a list that is **eq** to the tail of the original list starting at the first `a` at the top level. Thus, you could invoke **rplaca** on this returned value, provided that you first verify that member does not return **nil**.

See also **find** and **position**.

union *list1 list2* &key **:test :test-not :key** *[c]* Function
nunion *list1 list2* &key **:test :test-not :key** *[c]* Function

The **union** function returns a list representing the set that is the union of the sets represented by the arguments. Anything that is the element of at least one of the arguments is also an element of the result. If the **:key** functional argument is provided, then it is applied to elements of both arguments to select the portion to be compared. For example:

```
(union '(1 2 3) '(4 2 5)) => (1 2 3 4 5)
(union '(a b b c) '(a b d) => (a b b c d)
```

Generally, you can specify any predicate for **:test**, and the elements of the two lists are compared as follows: each element from the second list is tested against all the elements from the first list. If the two elements being tested are considered the same, one of the two is placed in the returned list.

If any element in either list does not match any element of the other list, the unmatched element appears in the result.

The **nunion** function operates the same as **union** but destructively alters all of the lists supplied as arguments by using their cons cells in building the returned list. Use this function only for its returned value, not for its side effects.

NOTE: These functions are different in Zetalisp mode—see Appendix A.

intersection *list1 list2* &key **:test :test-not :key** *[c]* Function
nintersection *list1 list2* &key **:test :test-not :key** *[c]* Function

The **intersection** function produces a list consisting of only those elements that are common to all the lists supplied as arguments. If neither argument has duplicates, then the result will not have duplicates. For example:

```
(intersection '(1 2 3) '(4 2 5)) => (2)
```

If the **:key** functional argument is provided, then it is applied to elements of both arguments to select the portion to be compared.

Generally, you can specify any predicate for **:test**, and the elements of the two lists are compared as follows: each element from the second list is tested against all the elements from the first list. If the two elements being tested are considered the same, one of the two is placed in the returned list. If any element in either list does not match any element of the other list, the unmatched element does not appear in the result.

The **nintersection** function operates the same as **intersection** but destructively alters *list1* by using its cons cells to build the returned list. Use these functions only for their returned values, not for their side effects.

NOTE: These functions are different in Zetalisp mode—see Appendix A.

---

adjoin *item list* &key :test :test-not :key *[c]* Function

This function adds *item* to the front of *list* if it is not already a member of *list*.

The keywords operate as described in paragraph 6.7, List Functions With Keyword Arguments. The default test is **eql**, and if the **:key** functional argument is provided, then it is applied to elements of *list* and to *item*. For example:

```
(adjoin 'steve '(mike john) => (steve mike john)
(adjoin 'steve '(steve mike john) => (steve mike john)
```

For any test specified (including the default), *item* is consed onto the front of *list* only if the test fails for every element of *list*.

See **pushnew** in paragraph 6.4, Functions Associated With Lists.

set-difference *list1 list2* &key :test :test-not :key *[c]* Function
nset-difference *list1 list2* &key :test :test-not :key *[c]* Function

The **set-difference** function returns a list containing all the elements of *list1* that do not match any element of *list2*. Neither list is destructively altered. If the :key functional argument is provided, then it is applied to elements of both list arguments.

The result contains no duplicate elements if *list1* contains none.

Any predicate can be used as the argument for **:test**, which compares the two lists as follows. Each element in *list1* is tested against every element in *list2*. The element from *list1* is placed in the returned list only if it fails the test for every element of *list2*. For example:

```
(set-difference '(1 2 3 4 5 6 7 8 9) '(1 2 3 5 7))
=> (4 6 8 9)
```

Note that the order of the output does not necessarily match the order of the input, and some elements of the output may share structure with the input.

The **nset-difference** function operates the same way as **set-difference** but destructively alters *list1*. Use these functions only for their returned values, not for their side effects.

set-exclusive-or *list1 list2* &key :test :test-not :key *[c]* Function
nset-exclusive-or *list1 list2* &key :test :test-not :key *[c]* Function

The **set-exclusive-or** function returns a list containing all the elements of *list1* that do not match any element of *list2* and all the elements of *list2* that do not match any element of *list1*. The result contains no duplicate elements if neither *list1* nor *list2* contains any. This operation is not destructive. If the :key functional argument is provided, it is applied to elements of both list arguments.

Note that the order of the output does not necessarily match the order of the input, and some elements of the output may share structure with the input.

The **nset-exclusive-or** operates the same as **set-exclusive-or** but destructively alters both *list1* and *list2*. Use these functions only for their returned values, not for their side effects.

subsetp *list1 list2* &key :test :test-not :key *[c]* Function

This function returns true if each element of *list1* is a member of *list2*. Otherwise, **subsetp** returns **nil**. If the **:key** functional argument is provided, then it is applied to elements of both list arguments.

Association Lists

6.8 The following functions provide a variety of operations for manipulating association lists. For a definition of association lists, see paragraph 6.1, List Definitions.

acons *key datum a-list* *[c]* Function

This function conses the association pair (*key* . *datum*) onto *a-list*. For example:

```
(acons :home "Austin" '((:name . "Bob") (:employer . "TI")))
=> ((:home . "Austin") (:name . "Bob") (:employer . "TI"))
```

copy-alist *list* *[c]* Function

This function copies the top level of association lists in the same manner as **copy-list** copies lists. Additionally, for every element in *list* that is a cons, **copy-alist** creates new cons cells that point to the same car and cdr elements.

pairlis *keys data* &optional *a-list* *[c]* Function

This function creates an association list from the *key* and *data* arguments. The two lists *keys* and *data* should be the same length. If *a-list* is specified, the created association list is consed onto it.

On the Explorer system, the new pairs appear in the returned value in the same order as they appear in the argument lists. For example:

```
(setq nums
  (pairlis '(one two) '(1 2) '((three . 3) (four . 4))))

nums => ((one . 1) (two . 2) (three . 3) (four . 4))
```

assoc *item a-list* &key :test :test-not :key *[c]* Function
assoc-if *predicate a-list* *[c]* Function
assoc-if-not *predicate a-list* *[c]* Function

The function **assoc** scans *a-list* for the first association pair whose key satisfies the argument for **:test** with *item*. If you specify the **:key** functional argument, then it is applied to each argument before the argument is passed to the test function. The returned value is the found association pair. For example:

```
(assoc 5 '((ace . hearts) (ace . clubs) (10 . diamonds)
             (5 . spades) (2 . spades)))
=> (5 . spades)

(assoc 'clubs '((ace . hearts) (6 . clubs))) => nil

(assoc 2 '((1 x y z) (2 b c d) (-10 a b c)))
=> (2 b c d)
```

If you want to update the associated value of an item in an *a-list*, use **setf** as in the following form:

```
(setf (rest (assoc item a-list)) new-value)
```

Consider the following example:

```
(setf standings '((Pittsburgh . 26-7) (New-York . 20-13)
                              (Saint-Louis . 10-23)))
(assoc 'Pittsburgh standings) => (Pittsburgh . 26-7)
(setf (rest (assoc 'Pittsburgh standings)) '27-7)
(assoc 'Pittsburgh standings) => (Pittsburgh . 27-7)
```

The **assoc-if** and **assoc-if-not** functions search and return the first association pair of *a-list* whose key satisfies or does not satisfy *predicate*. For example:

```
(setq pred 'numberp)
(setq alist '((x a) (2 b) (d c)))

(assoc-if pred alist) => (2 b)

(assoc-if #'numberp alist) => (2 b)

(assoc-if-not 'numberp alist) => (x a)
```

The specified *predicate* argument must follow the rules outlined in paragraph 6.7, List Functions With Keyword Arguments.

NOTE: These functions are different in Zetalisp mode—see Appendix A.

---

rassoc *item a-list* **&key :test :test-not :key** *[c]* Function
rassoc-if *predicate a-list* *[c]* Function
rassoc-if-not *predicate a-list* *[c]* Function

The **rassoc** function scans *a-list* for an association pair whose datum passes the test specified for **:test** and returns the found association pair. If you specify the **:key** functional argument, then it is applied to each argument before the argument is passed to the test function. For example:

```
(rassoc 'mild '((arizona . dry)(kansas . mild)
                          (minnesota . cold)))
=> (kansas . mild)
```

NOTE: These functions are different in Zetalisp mode—see Appendix A.

Property Lists

6.9 The following functions provide a variety of operations for manipulating property lists. For a definition of property lists, see paragraph 6.1, List Definitions.

getf *place indicator* &optional *default* *[c]* Function

This function is similar to **get** but differs in that generalized variables (not symbol names as in **get**) are used to reference a property list or part of a property list. Note that the **getf** function does not necessarily access a property list of a symbol; it accesses any location pointed to by the *place* argument. If this location is, for instance, a value cell, then **getf** treats whatever is in the value cell as a property list. The *indicator* argument is used to find the desired property. When the *default* argument is provided, its value is the returned value of the function if *indicator* is not contained in the property list of *place*; otherwise, **nil** is returned. The *place* argument has the same restrictions as for the *place* argument to **setf**. Consider the following examples:

```
(symbol-plist 'bar) => (one 1 two 2 three 3))          ; Current plist
(getf (symbol-plist 'bar) 'two) => 2                   ; Get from plist

(setf bar '(one 1 two 2 three 3))
(getf bar 'three) => 3                                 ; Get from value

(getf bar 'four :default-4) => :default-4             ; Get default
```

get-properties *place indicator-list* *[c]* Function

This function is similar to **getf** but takes a list of indicators (rather than a single indicator) as its second argument. Like **getf**, the **get-properties** function treats the value stored at *place* as a property list. This function looks for the first element in this property list whose indicator is also in *indicator-list*. The indicator must be **eq** to the property list item.

The **get-properties** function returns the following three values: the found indicator, the corresponding property value of the indicator, and the tail of the property list, starting with the found property value pair. The *place* argument has the same restrictions as for the *place* argument to **setf**. Consider the following example:

```
(symbol-plist 'foo) => (d 4 c 3 b 2 a 1)
(get-properties (symbol-plist 'foo) '(c b))
 => c
    3
    (c 3 b 2 a 1)
```

remf *place indicator* *[c]* Macro

This macro removes the property value whose indicator is *indicator* from the property list stored at *place*. The **eq** comparison is used to determine if *indicator* is in the property list indicated by *place*. If *indicator* is found in this property list, **remf** returns a true value; otherwise, it returns **nil**. The *place* argument has the same restrictions as for the *place* argument to **setf**. See also **remprop**.

List Predicates

6.10 The following functions can be used to test lists and conses.

consp *object* *[c]* Function

If *object* is a cons, this function returns true; otherwise, it returns **nil**. For example:

```
(consp '(black list)) => true
(consp 'list) => nil
(consp '( )) => nil
```

listp *object* *[c]* Function

If *object* is a list (including the empty list), this function returns true; otherwise, it returns **nil**. This predicate returns true even if *object* ends with a dotted pair. For example:

```
(listp '(black list)) => true
(listp 'list) => nil
(listp '( )) => true
(listp nil) => true
(listp '(1 . 2)) => true
```

NOTE: This function is different in Zetalisp mode—see Appendix A.

---

atom *object* *[c]* Function

This predicate returns true if *object* is not a cons; otherwise, it returns **nil**. Thus, `(atom ( ))` returns true because it is the same as `(atom nil)`.

endp *list* *[c]* Function

This function returns **nil** if *list* is a cons cell; it returns true if *list* is **nil**. This is the function Common Lisp recommends for terminating a loop that cdr's down a list.

tailp *sublist list* *[c]* Function

This predicate returns true if *sublist* is a sublist of *list* (that is, if *sublist* shares any cons cells with *list*); otherwise, it returns **nil**. Note that the following form always returns **nil**:

(tailp nil *any-list***)**

tree-equal *tree1 tree2* &key **:test :test-not** *[c]* Function

This function compares two trees recursively to all levels. Atoms must match according to the test specified by the **:test** functional argument (which defaults to **eql**). Conses must match recursively in both the car and the cdr.

If a **:test-not** functional argument is specified instead of **:test**, then two atoms match if the returned value of the **:test-not** function is **nil**.

null *object* *[c]* Function

This predicate returns true if its argument is **nil**; otherwise, it returns **nil**. This function is the same as **not** but is used for a different purpose: **null** indicates whether its argument is an empty list, whereas **not** is used to invert a logical value (such as testing to see if two objects are not equal). For example:

```
(null 0) => nil
(null '(black list)) => nil
(null ( )) => true
(null nil) => true
```

ARRAYS

7

Array Definitions

7.1 Objects of the **array** data type contain components arranged according to a rectilinear coordinate system. You can access the components in the array by specifying a list of numeric indices for each dimension of the array. Components of a *general array* can be any kind of Lisp objects, whereas *specialized arrays* are optimized to contain a single data type.

When an array is created, each dimension is given a size that is expressed as a nonnegative integer. Note that Common Lisp arrays have a zero origin; that is, the index for the first element in any dimension is 0, not 1. Therefore, the largest index permitted for a dimension is one less than the number specified for that dimension when the array was created. The smallest number for the index is always 0. The number of components contained in an array is the product of the sizes for all the dimensions. On the Explorer system, the size of an array is limited by only two constraints: the total number of components must be representable as a fixnum, and there must be sufficient virtual address space available.

The *rank* of an array is the number of its dimensions. On the Explorer system, the rank of an array must be less than 8, which is the value of **array-rank-limit**. If the rank of an array is 0, then it has no dimensions but is defined to contain one element. If any dimension of an array is 0 (which is not the same as having no dimensions), then it has all the associated properties of an array but has no components regardless of the rank.

Arrays can be created by the Reader using the Reader macro #*n*A(*array-elements*), where *n* is the rank of the array and *array-elements* is a set of nested lists, one level for each dimension in the array (see the **:initial-contents** option to **make-array** for more details). Each dimension of the array is determined by the length of the first sequence in that dimension. For example:

```
#2A((1 2)(3 4))                  ; Produces a 2 X 2 array.

#2A((1 2 3 4)(5 6 7 8)(9 0))     ; Produces a 3 X 4 array.
```

The printer also prints arrays using this format, where ***print-array*** is non-**nil**.

On the Explorer system, any generic sequence can be used instead of a list. Because character strings are sequences (see Section 9, Sequences), the following example places each character of the string into a separate cell of the array:

```
#2A("abc" "def")                 ; Produces a 2 X 3 array.
```

Several optional features for arrays can be used to enhance their power and flexibility. An *array leader* is like a one-dimensional array attached to the main array. The leader can be stored into and examined by a special set of functions different from those used for the main array. The leader is always one-dimensional and always can hold any kind of Lisp object, regardless of the type or rank of the main part of the array. Very often the main part of an array is used as a homogeneous set of objects, whereas the leader is used to remember a few associated nonhomogenous pieces of data. In this case, the leader is not used like an array. Each slot is used differently from the others. Do not use explicit numeric subscripts for the leader elements of such an array; instead, describe the leader with a **defstruct** using the **:array-leader** type option, and give each slot a name that describes its correspondence to the array leader. By convention, component 0 of the array leader contains the fill pointer (defined in the next paragraph). If you are not using a fill pointer, you should set slot 0 to **nil** or make sure that it contains a non-numeric value. If the main array is a non-Common Lisp named structure, then the name of the structure is kept in slot 1.

In one-dimensional arrays, a *fill pointer* can manage the linear allocation of the component slots. In arrays with fill pointers, the number of *active elements* grows until the last component is filled, but the physical size of the array does not change. To enable arrays to grow physically, you should declare them *adjustable* at the time you create them. Adjustable arrays can alter their size to be bigger or smaller dynamically.

Common Lisp defines that multidimensional arrays store their components in row-major order. In practice, this means that multidimensional arrays are stored as one-dimensional arrays. For example, suppose that you have created a 4-by-5 array. Since the array is stored in row-major order, elements (0,0) through (0,4) are stored in the first five memory slots allocated for the array (remember that Common Lisp arrays are zero-origined). Then, elements (1,0) through (1,4) are stored in the next five memory slots, and so on. In a 4-by-4-by-5 array, element (1,0,0) would be stored immediately after (0,3,4).

Row-major order provides a standard method of storage for array elements, enabling Common Lisp to define *displaced arrays* in which two arrays share some portion of their component set. For example, assume that *x* is a one-dimensional array and that *y* is a 2-by-2 array displaced into *x* with an offset of 1 (which means that the first element of *x* begins at the second element of *y*):

```
(aref x 1) <=> (aref y 0 0)
(aref x 2) <=> (aref y 0 1)
(aref x 3) <=> (aref y 1 0)
(aref x 4) <=> (aref y 1 1)
```

Arrays that do not have fill pointers, that are not displaced into other arrays, and that were not specified as adjustable when created are called *simple arrays*.

Vectors

7.1.1 One-dimensional arrays are defined to be of type **vector**, which is a subtype of **array**. Vectors are also of type **sequence**, as is anything of type **list**. Consequently, the functions in Section 9, Sequences, can also be used on vectors. A *string* is a specialized vector in which all components are of type **string-char**. Thus, functions in Section 8, Strings, apply to these specialized vectors. A *bit-vector* is a vector whose components are all of type **bit**; functions for manipulating bit-vectors are documented in this section.

A simple general vector can be created using one of the support functions (such as **make-array**) or by placing the data objects between the delimiters #(and). For example:

```
#(1492 nil 1776 1864 1985)        ; A vector of five components.
```

An Explorer extension allows you to optionally provide a length argument to the Reader macro: *#n(elements)*, where *n* specifies the length of the vector. If *n* is greater than the number of elements supplied, then the last element is repeated as many times as necessary to give the array *n* elements. For example:

```
#3(1 2)                           ; Equivalent to #(1 2 2)
```

A bit-vector is created by using the characters #* followed by 1s and 0s. For example, #*1100 creates a bit-vector of four components.

Internal Array Types

7.1.2 Common Lisp does not always explicitly define array types other than bit-arrays, strings, and general-purpose arrays. For the most part, Common Lisp defines array element types. The Explorer system explicitly defines array types that correspond to the internal array representation.

The following are the Explorer-defined array types (the prefix **art-** stands for *array type*):

- **art-q** — Arrays that can hold Lisp objects of all types.
- **art-q-list** — Arrays that can hold Lisp objects of all types and that can be handled as lists (except that the **rplacd** function cannot be used with these arrays).
- **art-fix** — Arrays that can contain any fixnum.
- **art-1b, art-2b, art-4b, art-8b, art-16b , art-32b** — Arrays that hold nonnegative integers and that store only the number of least-significant bits specified by their names. Thus, **art-1b** stores only the least-significant bit, **art-2b** stores only the two least-significant bits, and so on.
- **art-string** — Arrays that can contain elements of type **string-char.**
- **art-fat-string** — Arrays that hold characters consisting of an eight-bit code attribute and an eight-bit font attribute.
- **art-half-fix** — Arrays that contain halfword signed fixed numbers from -32768 to 32767.
- **art-single-float, art-double-float** — Arrays that contain floating-point numbers.
- **art-complex** — Arrays that contain any kind of number, including complex numbers.
- **art-complex-single-float, art-complex-double-float** — Arrays that contain real and complex numbers whose real and imaginary parts are both floating-point numbers.

Table 7-1 shows the correspondence between the Explorer extension array types and the Common Lisp array element types.

Table 7-1

Array Types and Array Element Types

Explorer Array Type (:type)	Common Lisp Array Element Type (:element-type)
`art-1b`	`bit`
`art-2b`	`(unsigned-byte 2)`
`art-4b`	`(unsigned-byte 4)`
`art-8b`	`(unsigned-byte 8)`
`art-16b`	`(unsigned-byte 16)`
`art-32b`	`(unsigned-byte 32)`
`art-fix`	`fixnum`
`art-half-fix`	`(signed-byte 16)`
`art-string`	`string-char`
`art-single-float`	`single-float`
`art-double-float`	`double-float`
`art-complex`	`complex`
`art-complex-single-float`	`(complex single-float)`
`art-complex-double-float`	`(complex double-float)`
`art-fat-string`	;No equivalent.
`art-q`	`t`
`art-q-list`	;No equivalent.
`art-reg-pdl`	;No equivalent.
`art-stack-group-head`	;No equivalent.
`art-special-pdl`	;No equivalent.

Note that **art-reg-pdl**, **art-stack-group-head**, and **art-special-pdl** are for internal use only.

Array Creation

7.2 The following functions are associated with creating arrays.

make-array *dimensions* &key **:element-type :initial-element :initial-contents :adjustable :fill-pointer :displaced-to :displaced-index-offset** *[c]* Function

make-array *dimensions* &key **:element-type :initial-element :initial-contents :adjustable :fill-pointer :displaced-to :displaced-index-offset :area :type :leader-length :leader-list :named-structure-symbol** Function

This function makes an array with the dimensions specified by *dimensions*, which should be a list of integers indicating the size of each dimension. The number of integers in *dimensions* equals the rank of the array. For one-dimensional arrays, you can simply specify an integer for *dimensions* rather than a list with one element. Two values are returned: the array itself and the number of words allocated to the array.

Every integer specified in *dimensions* must be less than the **array-dimensions-limit** constant. The total size of the array (that is, the product of its dimensions) must be less than the **array-total-size-limit** constant. If you specify an initial value of **nil** for *dimensions*, **make-array** makes a zero-dimensional array.

On the Explorer system, **make-array** has some additional keyword arguments that are considered extensions and may not be portable to other Common Lisp sites.

Common Lisp Standard Keywords

:element-type — The argument for **:element-type** must be a name that specifies the data type for the array elements. The default for **:element-type** is **t**, which means that the array's elements can be of any type. If you specify a type other than **t**, then all the elements subsequently stored in this array must be of the specified type. On the Explorer system, the internal array representation (see Table 7-1) that best matches the specified element type is used.

:initial-element — The argument for **:initial-element** specifies a single value to be stored in each element of the new array. If you do not provide an **:initial-element** argument (and do not provide either an **:initial-contents** or **:displaced-to** argument), the values of the array cells are undefined according to Common Lisp. As an Explorer extension, if the array type is numeric, the array is initialized to the appropriate form of 0; otherwise, the elements of the array are initialized to **nil**. You cannot use **:initial-element** and also use **:initial-contents** or **:displaced-to**.

:initial-contents — The argument to this keyword specifies a value for each element of the new array. This argument should be a sequence that has a length equal to the size of the first dimension. If the array has two dimensions, then each element of the original sequence should be a sequence equal in length to the size of the second dimension, and so on for as many dimensions needed. Recall that a sequence is either a list or a vector (and vectors include strings). If the array being created is zero-dimensional, then the value specified for **:initial-contents** becomes the single element in the array. For an array of any other dimensions, the argument for **:initial-contents** must be a sequence of sequences in which the number of elements in each list equals its corresponding dimension number. For example, note the creation of the following 5-by-2-by-3 array:

```
(make-array '(5 2 3) :initial-contents
   '(((Rick Rhoden P)       (14 9  2.72))
     ((Larry McWilliams P)  (12 11 2.93))
     ((John Tudor P)        (12 11 3.27))
     ((John Candelaria P)   (12 11 2.72))
     ((Jose DeLeon P)       (7 13 3.74)))
```

If you do not provide an **:initial-contents** argument (and do not provide either an **:initial-element** or **:displaced-to** argument), the values of the array cells are undefined according to Common Lisp. As an Explorer extension, if the array type is numeric, the array is initialized to the appropriate form of 0; otherwise, the elements of the array are initialized to **nil**. You cannot use **:initial-contents** and also use **:initial-element** or **:displaced-to**.

:adjustable — When a non-**nil** value is provided to this keyword, the array is *adjustable*, which means that it is permissible to change the array's size with the **adjust-array** function. The default value is **nil**. On the Explorer, all arrays are always adjustable; this argument is ignored.

:fill-pointer — The value supplied for this keyword is used to initialize the fill pointer index for the vector being created; that is, it defines the number of active elements in the newly created vector. The value should be an integer between 0 (inclusive) and the length of the array, or **t**. If you specify **t** for **:fill-pointer**, **make-array** uses the array's length for this option. The default value for this keyword is **nil**, meaning that there is no fill pointer; non-**nil** values are only permitted when creating vectors.

Using the **:fill-pointer** keyword is equivalent to using the **:leader-list** keyword with a list one component long.

:displaced-to — If this keyword is given a non-**nil** argument, a *displaced* array is constructed. To be compatible with Common Lisp, this value must be an array whose element type agrees with the type of array being created.

If the value for **:displaced-to** is an array, **make-array** creates an indirect array. On the Explorer system, if the value is an integer or a locative, **make-array** creates a regular displaced array that refers to the specified section of virtual address space.

If you use the **:displaced-to** option, you cannot specify a value for either the **:initial-element** or the **:initial-contents** option. Also note that the array being defined must not be larger than the array to which it is being mapped.

:displaced-index-offset — If this argument is specified, the value of the **:displaced-to** option should be an array. The value for **:displaced-index-offset** should be a nonnegative integer that is used as an index into the **:displaced-to** array. The element location indicated by this index becomes the first element in the displaced array.

The size of the array being defined plus the offset must not exceed the size of the array to which it is being mapped.

The following keyword arguments to **make-array** are Explorer extensions to the Common Lisp standard.

Explorer Extension Keywords

:area — This keyword specifies in which memory area (see paragraph 25.5, Storage and Allocation Areas) the array is to be created. It should be either an area number (an integer) or **nil** to indicate the default area.

:type — This is similar to the Common Lisp keyword **:element-type** but differs in that an Explorer array type name is used as its value (see Table 7-1). The default is **art-q**. The elements of the array are initialized according to the specified type: if the array is of a type whose elements can only be fixnums or floating-point numbers, then the array is automatically initialized to 0 or 0.0; otherwise, every element is initialized to **nil**.

:leader-length — If a corresponding value is given to this keyword, it must be a fixnum. The array then has an array leader. The length of the array leader is equal to this value, and the elements of the array leader are initialized to **nil** unless the **:leader-list** option (described below) is given a non-**nil** value.

:leader-list — If an argument value is given to this keyword, it must be a list. If the number of elements in the list is *n*, then the first *n* elements of the array leader are initialized from successive elements of this list. If the **:leader-length** keyword is not given a value, then the length of the array leader is *n*. If the **:leader-length** keyword is given a value and this value is greater than *n*, then the *n*th and following leader elements are initialized to **nil**. If the value specified for **:leader-length** is less than *n*, an error is signaled. The leader elements are filled in forward order; that is, the first element of the list is stored in leader element 0, the next element of the list is placed in in element 1 of the array leader, and so on.

:named-structure-symbol — The argument to this keyword is either **nil** or a symbol to be stored in the named-structure cell of the array. The array is tagged as a named structure (see the **:named** option to **defstruct** in paragraph 10.4.1, Common Lisp **defstruct** Options). If the array has a leader, then this symbol is stored in leader element 1 regardless of the value of the **:leader-list** keyword. If the array does not have a leader, then this symbol is stored in array element 0.

vector &rest *objects* *[c]* Function

This function constructs and returns a simple general vector (one-dimensional array) whose elements are objects. For example:

```
(setf pirate-starting-pitchers
     (vector "Rhoden" "McWilliams" "Tudor" "Candelaria" "DeLeon"))
=> #("Rhoden" "McWilliams" "Tudor" "Candelaria" "DeLeon")
```

Array Information

7.3 The following functions, constants, and variables are associated with retrieving information about array implementation and individual arrays.

array-dimension-limit *[c]* Constant

Any one dimension of an array must be smaller than the value of this constant. On the Explorer system, this constant is set to 16777214, the largest possible fixnum.

array-total-size-limit *[c]* Constant

The total number of elements in an array must be smaller than the value of this constant. On the Explorer system, this constant is set to 16777214, the largest possible fixnum.

array-rank-limit *[c]* Constant

The rank of an array must be smaller than this constant. On the Explorer system, this value is 8; therefore, arrays can have a rank between 0 and 7 (inclusive). All Common Lisp systems must have a rank limit of at least 8.

array-element-type *array* *[c]* Function

This function returns a type specifier that describes what kind of elements can be stored in *array* (see Section 12, Type Specifiers, for more information). Thus, if *array* is a string, the value is **string-char**. If *array* is an **art-1b** array, the value is **bit**. If *array* is an **art-2b** array, the value is `(mod 4)`. If *array* is an **art-q** array, the value is **t** (the type to which all objects belong).

array-type *array* Function

This function returns the Explorer array type name of *array*. For example:

```
(setq a (make-array '(3 5)))
(array-type a) => art-q
```

array-rank *array* *[c]* Function

This function returns the number of dimensions of *array*. This value is always a nonnegative integer less than the value of **array-rank-limit**.

array-dimension *array n* *[c]* Function

This function returns the length of dimension *n* of *array*. For example:

```
(setq a (make-array ´(2 3)))
(array-dimension a 0) => 2
(array-dimension a 1) => 3
```

array-dimensions *array* *[c]* Function

This function returns a list whose elements are the dimensions of *array*. For example:

```
(setq a (make-array ´(3 5)))
(array-dimensions a) => (3 5)
```

array-total-size *array* *[c]* Function
array-length *array* Function

These functions return *array*'s total number of components, which is the product of its dimensions. For example:

```
(setf pirate-starting-pitchers
   (make-array ´(5 2 3) :initial-contents
      ´(((Rick Rhoden P)          (14 9  2.72))
        ((Larry McWilliams P)   (12 11 2.93))
        ((John Tudor P)          (12 11 3.27))
        ((John Candelaria P)    (12 11 2.72))
        ((Jose DeLeon P)         (7 13 3.74)))

(array-total-size pirate-starting-pitchers) => 30
```

The **array-total-size** function ignores fill pointers in vector arrays that have them. The total size for a zero-dimensional array is always 1.

array-active-length *array* Function

If *array* has a fill pointer, it is returned; otherwise, the length of *array* is returned.

array-row-major-index *array* &rest *indices* *[c]* Function

This function calculates the cumulative index in *array* of the element at *indices*. Note the following equivalence:

```
(ar-1-force array (array-row-major-index array index1 index2 ...))
<=> (aref array index1 index2 ...)
```

array-element-size *array* Function

Given an array, this function returns the number of bits that fit in an element of that array. For arrays that can hold general Lisp objects, the result is 25, assuming you are storing fixnums in the array.

displaced-array-p *array* *[c]* Function

This function returns two values. If *array* is an indirect array, the first value is the array to which it is offset and the second value is the index to which it is offset. If *array* is not an indirect array, then the values **nil** and 0 are returned.

array-index-offset *array* Function

This function returns the index offset of *array* if it is an indirect array that has an index offset. Otherwise, it returns **nil**. The *array* argument can be any kind of array.

Accessing and Setting Arrays

7.4 The following functions are used to access arrays.

aref *array* &rest *subscripts* *[c]* Function

This function returns the element of *array* designated by the *subscripts*, which must be nonnegative integers and whose number must match the rank of *array*.

The **aref** function disregards fill pointers, unlike **elt**, which signals an error if an attempt is made to read past them.

To permanently change an array element, you can use **setf** with **aref**. For example:

```
(setf (aref pirate-starting-pitchers 3) 'Reuschel)
```

When dealing with multidimensional arrays, you frequently need to retrieve one of the array's elements by using a list of integers for the subscripts. You can do this easily using **apply**. As an example, suppose that array `rotation` is a 5-by-2-by-3 array and that you want to retrieve element `(4 1 0)` of this array using **apply**:

```
(setq x '(4 1 0))
(apply #'aref rotation x) => 7
```

The number of elements in this list must equal the rank of the array. This use of **apply** is also helpful for assigning or changing the value of a particular array element. For example:

```
(setf (apply #'aref rotation x) 8)
```

svref *simple-vector index* *[c]* Function

This is a special accessing function that operates on simple general vectors (vectors with no fill pointer, not displaced, and not adjustable).

ar-1-force *array index* Function
row-major-aref *array index* *[c]* Function

These functions access an array with a single subscript regardless of how many dimensions the array has. These functions can be useful for manipulating arrays of varying rank, as an alternative to maintaining and updating a list of subscripts or to creating one-dimensional indirect arrays. Note that you can update an item in an array by using **setf** with **ar-1-force** as a *place* argument. The **ar-1-force** function can also be used as an argument to **locf** to return the location of an item in the array.

Filling and Copying Arrays

7.5 The following functions are used for filling and copying arrays.

array-initialize *array value* &optional *start end* Function

This function stores *value* into all or part of *array*. Within this function, *array* is treated as a one-dimensional array regardless of its true rank. The *start* and *end* arguments are optional indices that delimit the part of *array* to be initialized. They should be nonnegative numbers smaller than the total size of the array (that is, they are not lists of indices). They default to the beginning and end of the array.

The **array-initialize** function is generally much faster than using a loop to assign each element.

fillarray *array x* Function

This function is obsolete; use the **fill** function instead (see Section 9, Sequences).

This function returns *array* or, if *array* is **nil**, the newly created array. There are two forms of this function, depending on the type of *x*. If *x* is a list, then **fillarray** fills up *array* with the elements of *list*. If *x* is too short to fill up all of *array*, then the last element of *x* is used to fill the remaining elements of *array*. If *x* is too long, the extra elements are ignored. If *x* is **nil** (the empty list), *array* is filled with the default initial value for its array type (**nil** or 0). If *x* is an array, then the elements of *array* are filled up from the elements of *x*. If *x* is too small, then the extra elements of *array* are not affected. The *array* argument can be any type of array. It can also be **nil**, in which case an array of type **art-q** is created. If *array* is multidimensional, the elements are accessed in row-major order: the last subscript varies most quickly. The same is true of *x* if it is a multidimensional array.

listarray *array* &optional *limit* Function

This function creates and returns a list whose elements are those of *array*. The *array* argument can be any type of array. If *limit* is specified, it should be a fixnum indicating how many elements from *array* to put in the returned list. Thus, the maximum length of the returned list is *limit*. If *array* is multidimensional, the elements are accessed in row-major order: the last subscript varies most quickly.

The **g-l-p** function is more efficient than **listarray**, when it is applicable.

g-l-p *array* Function

This function (which stands for *get list pointer*) returns a list that shares the storage of *array*. The *array* argument must be an **art-q-list** array. For example:

```
(setq a (make-array 4 :type 'art-q-list))
(aref a 0) => nil
(setq b (g-l-p a)) => (nil nil nil nil)
(setf (car b) t)
b => (t nil nil nil)
(aref a 0) => t
(setf (aref a 2) 30)
b => (t nil 30 nil)
```

list-array-leader *array* &optional *limit* Function

This function creates and returns a list whose elements are those of *array*'s leader. The *array* argument can be any type of array. If *limit* is specified, it should be a fixnum indicating how many elements from *array*'s leader are put in the returned list. Thus, the maximum length of the returned list is *limit*. If *array* has no leader, **nil** is returned.

copy-array-contents *from-array to-array* Function

This function copies the contents of *from-array* into the contents of *to-array*, component by component. The arguments *from-array* and *to-array* must be arrays. If *to-array* is shorter than *from-array*, the rest of *from-array* is ignored. If *from-array* is shorter than *to-array*, the rest of *to-array* is filled with **nil**, 0, or 0.0, according to the type of array. This function always returns **t**.

The entire length of *from-array* or *to-array* is used, ignoring any fill pointers. The leader itself is not copied.

The **copy-array-contents** function works on multidimensional arrays. If *from-array* is a string, then *to-array* contains char-code fixnums instead of string-chars. This function always returns **t**. The arguments *from-array* and *to-array* are treated as linear arrays, and components are taken in row-major order.

copy-array-contents-and-leader *from-array to-array* Function

This function is like **copy-array-contents** (described previously) but also copies the leader of *from-array* (if any) into *to-array*.

copy-array-portion *from-array from-start from-end to-array to-start to-end* Function

This function copies—component by component—the portion of the array *from-array*, with indices greater than or equal to *from-start* and less than *from-end*, into the portion of the array *to-array*, with indices greater than or equal to *to-start* and less than *to-end*. If there are more components in the selected portion of *to-array* than in the selected portion of *from-array*, the extra components are filled with the default value **nil**, 0, or 0.0, depending on the type of array. If there are more components in the selected portion of *from-array*, the extra components are ignored. Multidimensional arrays are treated the same way that **copy-array-contents** (described previously) treats them. If *from-array* is a string, then *to-array* contains **char-code** fixnums instead of **sdtring-chars**. This function always returns **t**.

bitblt *alu width height from-array from-x from-y to-array to-x to-y* Function

This function (which stands for *bit block transfer*) copies a rectangular portion of *from-array* into a rectangular portion of *to-array*. The value stored can be a Boolean function of the new value and the value already there, under the control of the function specified by the *alu* argument (see Table 3-4 and see the description for **boole**). The *from-array* and *to-array* arguments must be two-dimensional arrays of bits or bytes (**art-1b**, **art-2b**, **art-4b**, **art-8b**, **art-16b**, or **art-32b**). This function is most commonly used in connection with raster images for video displays.

The top-left corner of the source rectangle is (**aref** *from-array from-y from-x*). The top-left corner of the destination rectangle is (**aref** *to-array to-y to-x*). The *width* and *height* arguments are the dimensions of both rectangles. If *width* or *height* is 0, **bitblt** does nothing. The *x* coordinates and *width* are used as the second dimension of the array, because the horizontal index is the one that varies fastest in the screen buffer memory and the array's last index varies fastest in row-major order.

The *from-array* and *to-array* arguments can specify the same array. The **bitblt** function normally traverses the arrays in increasing order of *x* and *y* subscripts. If *width* is negative, then (**abs** *width*) is used as the width, but the processing of the *x* direction is performed backwards, starting with the highest value of *x* and working down. If *height* is negative, it is treated analogously. When you call **bitblt** on an array to itself, and when the two rectangles overlap, it may be necessary to work backwards to achieve effects such as shifting the entire array downwards by a certain number of rows. Note that if *width* or *height* is negative, the (*x*, *y*) coordinates specified by these arguments are not affected; these coordinates still specify the top-left corner even if **bitblt** starts at some other corner.

If the two arrays are of different types, **bitblt** works by bit and not by component. That is, if you invoke **bitblt** from an **art-2b** array into an **art-4b** array, then two components of the *from-array* correspond to one component of the *to-array*.

If **bitblt** goes outside the bounds of the source array, the copying wraps around to the beginning of the source array. This feature allows such operations as the replication of a small stipple pattern through a large array. If **bitblt** goes outside the bounds of the destination array, it signals an error.

If `src` is a component of the source rectangle and `dst` is the corresponding component of the destination rectangle, then **bitblt** changes the value of `dst` to `(boole alu src dst)`. See the **boole** function in paragraph 3.9, Number Component Extraction. There are symbolic names for some of the most useful *alu* functions; they are **boole-1** (plain copy), **boole-ior** (inclusive or), **boole-xor** (exclusive or), and **boole-andc1** (and-with-complement of source).

The **bitblt** function is written in highly optimized microcode and performs much faster than would the same function written with ordinary **aref** and **aset** operations. Unfortunately, this optimization causes **bitblt** to have a couple of strange restrictions. Wraparound does not work correctly if *from-array* is an indirect array with an index offset. The **bitblt** function signals an error if the second dimensions of *from-array* and *to-array* are not both integral multiples of the machine word length. For **art-1b** arrays, the second dimension must be a multiple of 32. For **art-2b** arrays, it must be a multiple of 16, and so on.

Bit-Vectors and Bit-Arrays

7.6 An array that contains only bits is called a *bit-array*, and a vector that contains only bits is called a *bit-vector*. The default method for printing a bit-vector uses a symbolic representation. For example:

```
#*1010 => #<art-1b-4>
```

When the global variable ***print-array*** is set to true, the printed representation displays the contents of the arrays. For example:

```
#*1010 => #*1010
```

For the sake of clarity, the latter notation is used in the examples on this topic.

The following functions are used for manipulating bit-vectors and bit-arrays.

bit *bit-array* &rest *subscripts* *[c]* Function
sbit *simple-bit-array* &rest *subscripts* *[c]* Function

These functions are special accessing functions defined to work only on bit-vectors and only on simple bit-vectors, respectively.

As with **aref**, you can use **setf** with **bit** or **sbit** to change the contents of a bit-array cell permanently.

bit-and *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function
bit-ior *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function
bit-xor *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function
bit-eqv *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function
bit-nand *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function
bit-nor *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function
bit-andc1 *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function
bit-andc2 *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function
bit-orc1 *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function
bit-orc2 *bit-array1 bit-array2* &optional *result-bit-array* *[c]* Function

These functions perform their respective Boolean operations component by component on bit-arrays. For each of these functions, *bit-array1* and *bit-array2* must match in size and shape, and all of their components must be integers. Corresponding components of *bit-array1* and *bit-array2* are taken and passed to one of **logand**, **logior**, and so on, to process a component of the result array.

If *result-bit-array* is **t**, the results are stored in *bit-array1*. If *result-bit-array* is not **t** but is non-**nil**, then this array is assumed to be another array, into which the results are stored. Otherwise, a new array of the same type as *bit-array1* is created and used for the result. In any case, the value returned is the array where the results are stored.

In Common Lisp, these functions were introduced for bit-arrays only. On the Explorer, these functions accept not only bit-arrays but any array whose components are integers.

Table 7-2 indicates the result of one component of a bit-array when these functions are applied to two-argument bit-arrays.

Table 7-2 Bitwise Logical Operations on Bit-Arrays

Function Name					Logical Operation
bit-array1	**0**	**0**	**1**	**1**	
bit-array2	**0**	**1**	**0**	**1**	
bit-and	0	0	0	1	And
bit-ior	0	1	1	1	Inclusive or
bit-xor	0	1	1	0	Exclusive or
bit-eqv	1	0	0	1	Exclusive nor
bit-nand	1	1	1	0	Nand
bit-nor	1	0	0	0	Nor
bit-andc1	0	1	0	0	And the complement of *bit-array1* with *bit-array2*
bit-andc2	0	0	1	0	And *bit-array1* with complement of *bit-array2*
bit-orc1	1	1	0	1	Or complement of *bit-array1* with *bit-array2*
bit-orc2	1	0	1	1	Or *bit-array1* with complement of *bit-array2*

Consider the following example:

```
(bit-and #*0110 #*1100) => #*0100
(bit-ior #*0110 #*1100) => #*1110
(bit-xor #*0110 #*1100) => #*1010
(bit-andc1 #*0110 #*1100) => #*1000
(bit-orc1 #*0110 #*1100) => #*1101
```

bit-not *bit-array* &optional *result-bit-array* *[c]* Function

This function performs **lognot** on each component of *bit-array* to produce a component of the result. If *result-bit-array* is non-**nil**, the result components are stored in it (*result-bit-array* must match *bit-array* in size and shape). Otherwise, a new array of the same type as *bit-array* is created and used to hold the result. The returned value of **bit-not** is the array where the results are stored. Consider the following example:

```
(setq x #*1010)
(setq y #*0011)

; Copy of x inverted returned.
(bit-not x) => #*0101

; Change y to inverted x.
(bit-not x y) => #0101
y => #*0101
x => #*1010

; Invert x.
(bit-not x x) => #*0101
x => #*0101
```

Fill Pointers and Array Leaders

7.7 The following functions manipulate fill pointers and array leaders.

NOTE: Although fill pointers are part of the Common Lisp specification, array *leaders* are not. On the Explorer system, however, fill pointers are implemented using array leaders. To ensure that your programs are portable Common Lisp, use the Common Lisp functions.

An array leader is like a one-dimensional **art-q** array attached to the main array and is used to store a few nonhomogenous pieces of information (see paragraph 7.1, Array Definitions, for more details on fill pointers and array leaders). By convention, element 0 of the array leader holds the number of active array elements and is called the *fill pointer*. Element 1 is used in conjunction with the named structure feature described in Section 10, Structures, to associate a data type with the array.

fill-pointer *vector* *[c]* Function

This function returns the fill pointer of *vector*. An error is signaled if *vector* does not have a fill pointer. This function can be used with **setf** to set the array's fill pointer.

vector-push *item array* *[c]* Function

If *array* is not already full, this function first stores *item* in the cell of *array* specified by the fill pointer of *array* and then increments the fill pointer by 1. The *array* argument must be a vector with a fill pointer, and *item* can be any object that can be stored in the array. The returned value is the original value of the fill pointer (that is, before it is incremented). As a safety-locking feature, the array is referenced and the fill pointer is incremented without interruption. If *array* is already full, **vector-push** returns **nil**, and the fill pointer for *array* is unchanged.

vector-push-extend *item array* &optional *extension* *[c]* Function

Like **vector-push**, this function pushes *item* onto *array*. However, if *array* is already full, **vector-push-extend** extends the size of *array* to accommodate *item*. In this case, the optional *extension* argument, if provided, specifies the number of cells to be added to *array*.

On the Explorer system, this value generally defaults to 64 or to one-fourth of the size of the array, whichever is larger.

vector-pop *array* *[c]* Function

This function decreases the fill pointer by 1 and returns the array element designated by the new value of the fill pointer. The *array* argument must be a vector with a fill pointer. The two operations (decrementing and array referencing) proceed without interruption. If there are no more elements to pop (the fill pointer has already reached 0), an error is signaled.

array-leader *array index* Function

This function returns the element specified by *index* from *array*'s leader. The *array* argument should be an array with a leader, and *index* should be a fixnum. This function is analogous to **aref**. It can also be used as a *place* argument for **setf**.

store-array-leader *item array index* Function

This function stores the value for *x* in the element specified by *index* from *array*'s leader and returns *item*. This is analogous to **aset**. It is preferable to use **setf** with **array-leader** as a generalized variable. The *array* argument should be an array with a leader, and *index* should be an integer. The argument *item* can be any object.

array-leader-length *array* Function

This function returns the length of *array*'s leader if it has one, or **nil** if it does not.

Modifying Array Characteristics

7.8 The following function is used to change the dimensions of an already created array.

adjust-array *array new-dimensions* &key **:initial-element** **:element-type** **:initial-contents** **:fill-pointer** **:displaced-to** **:displaced-index-offset** *[c]* Function

This function modifies various aspects of an array. The argument *array* is modified in its current location if possible; otherwise, a new array is created and subsequent references are forwarded to it in a transparent way. In either case, the adjusted array is returned. The arguments have the same names as arguments to **make-array** and signify approximately the same thing. However, note the following individual cases.

new-dimensions — You can change the dimensions of the array, but you cannot change the rank of the array.

:initial-element — If this keyword is specified, then all newly created locations in the array are initialized to this value. If the newly adjusted array is a displaced array (if the **:displaced-to** option is used), then **:initial-element** has no effect.

:element-type — This keyword is merely an error check; **adjust-array** cannot change the array type. If the array type of *array* is not what **:element-type** implies, an error is signaled.

:initial-contents — If this keyword is specified, then the contents of the adjusted array are initialized as with **make-array**. None of the old contents of the array are accessible in the newly adjusted array. As with **make-array**, you cannot use both **:initial-contents** and **:initial-element**. If the newly adjusted array is a displaced array (if the **:displaced-to** option is used), then **:initial-contents** has no effect.

:fill-pointer — If this keyword is specified, it is used as the new fill pointer in the adjusted array. Otherwise, the adjusted array has a leader with the same contents as in the original array. If data is copied from the old array to a new adjusted array location, neither the old fill pointer nor a newly specified fill pointer is used to limit the amount of data copied (**array-total-size** is used).

:displaced-to — If this keyword is specified, then the newly adjusted array is displaced as indicated by the **:displaced-to** and **:displaced-index-offset** keywords. These arguments work the same way as in **make-array**.

:displaced-index-offset — If this keyword is specified, it works in conjunction with **:displaced-to** the same as with **make-array**. If the old array is currently displaced, you should note that the default for this keyword is still 0 and not the offset value of the old array.

According to Common Lisp, an array's dimensions can be adjusted only if the **:adjustable** option is specified to **make-array** with a non-**nil** value when the array is created. The Explorer system does not distinguish adjustable and nonadjustable arrays; any array can be adjusted. Portable Common Lisp programs should not call **adjust-array** on an array that was not created with the **:adjustable** option.

For example, suppose you set the variable `players` to the following 5-by-4 array:

```
#2A((Kirk     Gibson  outfielder   Tigers)
    (Darrell  Porter  catcher      Cardinals)
    (Donnie   Moore   pitcher      Angels)
    (Cesar    Cedeno  outfielder   Cardinals)
    (Dave     Kingman designated-hitter  Athletics))
```

Then you call **adjust-array** on `players` with the following arguments:

```
(adjust-array players '(3 5) :initial-element 'free-agent)
```

The following array is returned by **adjust-array**:

```
#2A((Kirk     Gibson outfielder   Tigers     free-agent)
    (Darrell  Porter catcher      Cardinals  free-agent)
    (Donnie   Moore  pitcher      Angels     free-agent)
```

If some array `ary1` has been displaced to another array `ary2` that is subsequently used as an argument to **adjust-array**, the displacement is unaffected. That is, `ary1` is still displaced to `ary2`. However, because `ary2` has been adjusted, `ary1` is also adjusted relative to the adjustment made to `ary2`. For example:

```
(setf ary2 (make-array '(2 2) :adjustable t))
(setf ary1 (make-array 4 :displaced-to ary2))
(setf (aref ary1 0) 0 (aref ary1 1) 1 (aref ary1 2) 2 (aref ary1 3) 3)
```

The following table shows the memory locations, indices, and contents of the two arrays before adjustment:

Location	ary2	ary1	Contents
0	(0,0)	(0)	0
1	(0,1)	(1)	1
2	(1,0)	(2)	2
3	(1,1)	(3)	3

Next, `ary2` is adjusted as follows:

```
(setf ary2 (adjust-array ary2 '(3 3)))
```

The following table shows the memory locations, indices, and contents of the two arrays after adjustment:

Location	ary2	ary1	Contents
0	(0,0)	(0)	0
1	(0,1)	(1)	1
2	(0,2)	(2)	**nil**
3	(1,0)	(3)	2
4	(1,1)		3
5	(1,2)		**nil**
6	(2,0)		**nil**
7	(2,1)		**nil**
8	(2,2)		**nil**

Note that the contents of `ary1` have changed such that `(aref ary1 2)` now returns **nil** to match the corresponding new element of `ary2` (0,2) and that `ary1` no longer holds the value 3 because none of its elements correspond to element (1,1) of `ary2`.

sys:change-indirect-array *array type dimension-list displaced-p index-offset* Function

This function changes the type, size, or target pointed at for the indirect array specified by *array*. The *type* argument specifies the new array type. The *dimension-list* argument specifies the new dimensions of the array. The *displaced-p* argument specifies the target that *array* should point to (an array, locative, or fixnum). The *index-offset* argument specifies the new offset in the new target.

Array Predicates

7.9 The following functions are predicates used to perform various tests on arrays.

arrayp *object* *[c]* Function

This predicate returns true if *object* is an array; otherwise, it returns **nil**.

vectorp *object* *[c]* Function

This predicate returns true if *object* is an array of rank 1.

simple-vector-p *object* *[c]* Function

This predicate returns true if *object* is an array of rank 1 that has no fill pointer, that is not displaced, and that can hold any Lisp object as an element.

bit-vector-p *object* *[c]* Function

This predicate returns true if *object* is an array of rank 1 that allows only 0 and 1 as elements.

simple-bit-vector-p *object* *[c]* Function

This predicate returns true if *object* is an array of rank 1 that has no fill pointer, that is not displaced, and that allows only 0 and 1 as elements.

array-in-bounds-p *array* &rest *subscripts* *[c]* Function

This predicate returns true if *subscripts* are legitimate subscripts for *array*; otherwise, it returns **nil**. Note that this predicate does not observe fill pointers.

adjustable-array-p *array* *[c]* Function

This predicate returns true if *array* can be adjusted with **adjust-array** (that is, if the **:adjustable** keyword was specified when *array* was made with **make-array**). On the Explorer system, this function always returns true because all arrays are adjustable.

array-has-fill-pointer-p *array* *[c]* Function

This predicate returns true if *array* has a fill pointer. On the Explorer, system, the array specified by *array* must have a leader, and leader element 0 must be an integer. While array leaders are not standard to Common Lisp, fill pointers are, and so is this function.

array-displaced-p *array* Function

This predicate returns true if *array* is any kind of displaced array (including an indirect array). Otherwise, it returns **nil**. The argument *array* can be any kind of array.

array-indirect-p *array* Function

This predicate returns true if *array* is an indirect array. Otherwise, it returns **nil**. The *array* argument can be any kind of array.

array-indexed-p *array* Function

This predicate returns true if *array* is an indirect array with an index offset. Otherwise, it returns **nil**. The *array* argument can be any kind of array.

array-has-leader-p *array* Function

This predicate returns true if *array* has a leader; otherwise, it returns **nil**.

Matrices and Systems of Linear Equations

7.10 The following functions are used to perform operations on matrices.

math:multiply-matrices *matrix-1 matrix-2* &optional *matrix-3* Function

This function multiplies *matrix-1* by *matrix-2*. If *matrix-3* is supplied, **multiply-matrices** stores the results in *matrix-3* and returns *matrix-3*, which should be of exactly the proper dimensions for containing the result of the multiplication; otherwise, this function creates an array to contain the answer and returns this array. All matrices must be either one-dimensional or two-dimensional arrays, and the first dimension of *matrix-2* must equal the second dimension of *matrix-1*.

math:invert-matrix *matrix* &optional *into-matrix* Function

This function computes the inverse of *matrix*. If *into-matrix* is supplied, this function stores the result in *into-matrix* and returns *into-matrix*; otherwise, **invert-matrix** creates an array to hold the result and returns this array. The argument *matrix* must be a two-dimensional square array. The Gauss-Jordan algorithm with partial pivoting is used. Note that if you want to solve a set of simultaneous equations, you should not use this function; use **math:decompose** and **math:solve**.

math:transpose-matrix *matrix* &optional *into-matrix* Function

This function transposes *matrix*. If *into-matrix* is supplied, this function stores the result in *into-matrix* and returns *into-matrix*; otherwise, **transpose-matrix** creates an array to hold the result and returns this array. The *matrix* argument must be a two-dimensional array. The *into-matrix* argument, if provided, must be two-dimensional and have exactly the proper dimensions to hold the transposition of *matrix*.

math:list-2d-array *array* Function

This function returns a list of lists containing the values in *array*, which must be a two-dimensional array. There is one element for each row; each element is a list of the values in that row.

math:fill-2d-array *array list* Function

This function fills a two-dimensional array and is thus the complement of **math:list-2d-array**. The *list* argument should be a list of lists, with each element being a list corresponding to a row. If *list* is not long enough, **math:fill-2d-array** wraps around, starting over at the beginning. The lists that are elements of *list* also wrap around if more elements are needed.

math:determinant *matrix* Function

This function returns the determinant of *matrix*. The *matrix* argument must be a two-dimensional square matrix.

The next two functions are used to solve sets of simultaneous linear equations. The **math:decompose** function takes a matrix holding the coefficients of the equations and produces the LU decomposition; this decomposition can then be passed to **math:solve** along with a vector of right-hand sides to get the values of the variables. If you want to solve the same equations for many different sets of right-hand side values, you need only call **math:decompose** once. In terms of the argument names used in the following descriptions, these two functions exist to solve the vector equation $Ax=b$ for x. A is a matrix. The values b and x are vectors.

math:decompose *a* &optional *lu ps* Function

This function computes the *lu* decomposition of matrix *a*. If the array specified by *lu* is non-**nil**, this function stores the result in *lu* and returns *lu*; otherwise, **decompose** creates an array to hold the result and returns this array. The lower triangle of *lu*, with 1s added along the diagonal, is L, and the upper triangle of *lu* is U such that the product of L and U is *a*. Gaussian elimination with partial pivoting is used. The *lu* array is permuted by rows according to the permutation array *ps*, which is also produced by **decompose**; if the argument *ps* is supplied, the permutation array is stored into it; otherwise, an array is created to hold it. This function returns two values: the LU decomposition and the permutation array.

math:solve *lu ps b* &optional *x* Function

This function takes the *lu* decomposition and associated permutation array produced by **math:decompose** and solves the set of simultaneous equations defined by the original matrix *a* given to **math:decompose** and the right-hand sides in the vector *b*. If *x* is supplied, the solutions are stored into *x* and *x* is returned; otherwise, an array is created to hold the solutions and this array is returned. The argument *b* must be a one-dimensional array.

Planes

7.11 The following functions are used for manipulating planes. A *plane* is an array whose bounds in each dimension are plus infinity and minus infinity; thus, all integers are legal as indices. Planes can be of any rank. When you create a plane, you need not specify the size, just the rank. You also must specify a default value to which every component of the plane is initialized at the time of creation. Because you can never change more than a finite number of components, only a finite region of the plane must be stored. When you refer to an element for which space has not yet been allocated, you simply receive the default initialization value.

make-plane *rank* &key :type :default-value :extension :initial-dimensions :initial-origins Function

This function creates and returns a plane. The *rank* argument specifies the number of dimensions. The keyword arguments are as follows.

:type — The array type symbol (for example, **art-1b**) specifying the type of array out of which the plane is made.

:default-value — The value to which each plane component is initialized when the plane is created.

:extension — The amount by which to extend the plane, in any direction, when **plane-store** is invoked outside of the currently stored portion.

:initial-dimensions — The value **nil** or a list of integers whose length is *rank*. If this keyword is not **nil**, each element corresponds to one dimension, specifying the initial width in that dimension to allocate for the array.

:initial-origins — The value **nil** or a list of integers whose length is *rank*. If not **nil**, each element corresponds to one dimension, specifying the smallest index in that dimension for which storage should initially be allocated.

For example:

```
(make-plane 2 :type 'art-4b :default-value 3)
```

This example creates a two-dimensional plane of type `art-4b` with default value 3.

plane-origin *plane* Function

This function returns a list of numbers (subscript indices), giving the lowest coordinate values actually stored in *plane*.

plane-default *plane* Function

This function returns the default value to which each plane component was initialized when *plane* was created.

plane-extension *plane* Function

This function returns the amount by which to extend *plane*, in any direction, when **plane-store** is invoked outside of the currently stored portion.

plane-aref *plane* &rest *subscripts* Function
plane-ref *plane subscripts* Function

These two functions return the contents of a specified component of *plane*. The *subscripts* argument specifies the element to be returned. These functions differ only in the way they take their arguments: **plane-aref** accepts the subscripts as arguments, while **plane-ref** accepts a list of subscripts.

plane-aset *datum plane* &rest *subscripts* Function
plane-store *datum plane subscripts* Function

These two functions store *datum* into the specified element of plane, extending it if necessary, and return *datum*. The *subscripts* argument specifies the component to be accessed. These functions differ only in the way they take their arguments: **plane-aset** accepts the subscripts as arguments, while **plane-store** accepts a list of subscripts.

8

STRINGS

String Definitions

8.1 The data type **string** defines an object which is a one-dimensional array whose elements are of type **string-char**. *String-chars* are defined as character objects whose bit and font attributes are both set to 0. Therefore, strings are specialized vectors. Because they are vectors, strings are also of type **sequence**. Thus, operations described in Section 9, Sequences, also accept strings as arguments.

The written syntax for a string is simply a sequence of characters preceded by and terminated by double quotation marks ("). The Lisp Reader does not attempt to map lowercase characters to uppercase while reading a string sequence. If you want to include either a double quotation mark or a backslash (\) character in a string, you must precede the character by a backslash. As with symbol names, the preceding backslash does not become part of the name but serves only as a signal to the Lisp Reader to interpret the next character as part of the sequence. Note that during the reading of strings, the vertical bar character (|) has no special meaning as it does when symbol names are read. Consider the following examples:

```
(length "") => 0
(length "abcdefghijklmnopqrstuvwxyz") => 26
(length "./\.") => 3   ;Note that the backslash does not become part
                       ;of the string.
```

On the Explorer system, string-char objects are implemented with eight-bit code attributes ranging from 0 to 255 and with no font or bit attributes, since these are defined to be 0. An alternative array type called **art-fat-string** allows an additional eight bits to be allocated for each character. These bits can be used to save font information or represent an extended character code attribute set. However, the **art-fat-string** type is not part of the Common Lisp standard and therefore may not be portable. On the Explorer system, fat-strings can be used as string arguments to Common Lisp functions. Note that those functions using **char=** as a test condition will test for equality in the font field.

Functions described in this section whose names begin with the **string-** prefix accept symbols for their string arguments. For such functions, the print name of the symbol is used. Because print names cannot be modified, symbols cannot be used as arguments to functions that attempt to modify their string arguments. Note that the generic sequence operations do not accept symbols as sequence arguments.

The characters of a string are stored in order, with the leftmost character located at index 0 of the vector. You can access the contents of a string in the same way that you access the contents of an ordinary array:

```
(setf EXPLORER "Ai") => "Ai"
(aref EXPLORER 1) => #\i
(setf (aref EXPLORER 1) #\I) => #\I
EXPLORER => "AI"
```

If you create a string that does have a fill pointer, string operations generally operate only on the active portion of the vector. Although these array access functions always operate properly on strings, Common Lisp defines two special routines (**char** and **schar**) for accessing the elements of a string. In some implementations, these routines provide an optimized access algorithm. On the Explorer, however, **char** and **schar** are equivalent to **aref**.

Note that the **read** function is defined by Common Lisp always to create a simple vector, that is, one that does not have a fill pointer, is not adjustable, and is not offset into another array. A string with these qualities is called a *simple string*. On the Explorer system, however, all arrays are adjustable.

Print routines can generate output in two different forms. Some routines print the string as interpreted by the Lisp Reader (that is, with escape characters removed), and others print the string in a syntax suitable for input to the Lisp Reader (that is, with escape characters in appropriate places so that the string can be parsed by the Reader again). Thus, you probably should not include escape characters in documents such as text reports, but do use escape character encoding for data that will subsequently be reread by the Lisp Reader. See the *Explorer Input/Output Reference* for more details on print routines.

Character Access in Strings

8.2 The following functions are associated with accessing individual characters in strings.

char *string index* *[c]* Function
schar *simple-string index* *[c]* Function

These functions are used for accessing individual characters from *string*. They return the character at the position specified by the argument *index*, which must be a nonnegative integer less than the length of *string*. Note that the string specified for **char** can include a fill pointer but that the string specified for **schar** cannot—it must be a simple string (a string without a fill pointer; see paragraph 7.1, Array Definitions, for a description of fill pointers). As with all sequences in Common Lisp, indexing for these functions is zero-origin based. Consider the following examples:

```
(char "AbCdEfGhIjKlMnOpQrStUvWxYz" 0) => #\A
(schar "AbCdEfGhIjKlMnOpQrStUvWxYz" 1) => #\b
```

The **setf** macro can be used with **char** or **schar** to destructively replace a character within a string. On the Explorer system, **char** and **schar** are synonymous with the Common Lisp version of **aref**, but on other implementations, **char** and **schar** may be more efficient.

String Equality

8.3 The following functions are associated with determining whether strings are equal.

When you use the start and end keywords as indices into the string arguments, comparisons start with the character indexed by the value given to **:start1** and/or **:start2**, and continue up to but do not not include the character indexed by the value given to **:end1** and/or **:end2**. The default for the start keyword is 0, and the default for the end keyword is the active length of the string. Thus, the default case uses all of the data in the string.

string= *string1 string2* &key **:start1 :start2 :end1 :end2** *[c]* Function

This function compares two strings, returning true if they are equal according to **char=** and **nil** if they are not. The character case and font of the arguments' characters is taken into account during the comparison. For example:

```
(string= "XYZA" "xyza") => nil
(string= "xyza" "xyza") => true
(string= "abcd" "abce") => nil
(string= "coffee" "feed" :start1 3 :end2 3) => true
```

When **equal** compares string arguments, it uses **string=** to make the comparison.

string/= *string1 string2* &key **:start1 :end1 :start2 :end2** *[c]* Function

This function returns a number if the specified portions of *string1* and *string2* are different. The number returned is actually the index, relative to *string1*, of the first difference between the strings. Case is significant in comparing characters. For example:

```
(string/= "abcde" "abcde") => nil
(string/= "abcde" "abdef") => 2
(string/= "abcde" "aBcde") => 1
```

Lexicographical Comparison

8.4 The following functions are associated with the lexicographical comparison of strings, that is, a sorted ordering for strings including distinctions between uppercase and lowercase characters.

string< *string1 string2* &key **:start1 :end1 :start2 :end2** *[c]* Function
string> *string1 string2* &key **:start1 :end1 :start2 :end2** *[c]* Function
string<= *string1 string2* &key **:start1 :end1 :start2 :end2** *[c]* Function
string>= *string1 string2* &key **:start1 :end1 :start2 :end2** *[c]* Function

These functions compare all or the specified portions of *string1* and *string2* using lexicographic order. Characters are compared using **char<** and **char=** so that font and alphabetic case are taken into account.

These functions operate in the following way:

1. Identify the string or substring to be operated on for both *string1* and *string2*.

2. Compare corresponding elements of the strings:

   - If each of the corresponding characters is **char=**, then if the function call was **string<=** or **string>=**, return the index of *string1* that is one past the last character tested. If equal strings were not allowed (that is, if **string<** or **string>** was called), then return **nil**.

   - If any two characters are not **char=**, then the appropriate inequality test (**char>** if **string>** was called, or **char<** if **string<** was called) is performed. If the result of this test is true, the index of the character in *string1* is returned. If the result of this test is not true, **nil** is returned.

Note that Common Lisp specifies that all uppercase letters will collate correctly, that all lowercase letters will collate correctly, and that digits 0 through 9 will collate correctly. However, it does not specify how a mixture of numbers, uppercase letters, and lowercase letters will collate. Thus, the letter *A* may be greater than the letter *a*, or the letter *a* may be greater than the letter *A*. On the Explorer, all uppercase letters are less than their corresponding lowercase letters.

Consider the following examples:

```
(string< "AB1CD" "AB2CD") => 2
(string< "AB2CD" "AB1CD") => nil
(string>= "ABCD" "ABCD") => 4
(string>= "THIS" "WHICH, THAT ONE?" :end1 3 :start2 7 :end2 10) => 2
```

string-compare *string1 string2* &optional *start1 start2 end1 end2* Function

This function compares all or the specified portions of two strings using lexicographical order (as defined by **char-lessp**). The arguments are interpreted the same way as in **string=** except that the arguments are positional rather than associated with keywords. The result is 0 if the strings are equal, a negative number if *string1* is less than *string2*, or a positive number if *string1* is greater than *string2*. If the strings are not equal, the absolute value of the number returned is one greater than the index (in *string1*) where the first difference occurred.

String Comparison Ignoring Case

8.5 For the following functions associated with lexicographical comparison of strings, distinctions between uppercase and lowercase characters are ignored.

string-equal *string1 string2* &key :start1 :start2 :end1 :end2 *[c]* Function

This function compares two strings, returning true if they are equal according to **char-equal** and **nil** if they are not. Unlike **string=**, **string-equal** is not character case and font sensitive.

The arguments to the keywords **:start1** and **:start2** are the starting indices into the strings. The arguments to the keywords **:end1** and **:end2** are the final indices; the comparison stops just *before* the final index. The default value for the start keywords is 0; for the end keywords, the default is **nil**. If no argument or **nil** is provided to the end keywords, then the comparison stops at the end of the string. If the two strings are of unequal length, **string-equal** returns false. Consider the following examples:

```
(string-equal "Match" "match") => true
(string-equal "match" "Match") => true
(string-equal "miss" "match") => nil
(string-equal "element" "select"
  :start1 0 :end1 1 :start2 3 :end2 4) = > true
```

string-not-equal *string1 string2* &key :start1 :end1 :start2 :end2 *[c]* Function

This function returns a number if the specified portions of *string1* and *string2* are different. The number returned is actually the index, relative to *string1*, of the first difference between the strings. Case is significant in comparing characters. For example:

```
(string-not-equal "abcde" "abcde") => nil
(string-not-equal "abcde" "ABCDE") => nil
(string-not-equal "abcde" "abdef") => 2
```

string-lessp *string1 string2* &key **:start1 :end1 :start2 :end2** *[c]* Function
string-greaterp *string1 string2* &key **:start1 :end1 :start2 :end2** *[c]* Function
string-not-greaterp *string1 string2* &key **:start1 :end1 :start2 :end2** *[c]* Function
string-not-lessp *string1 string2* &key **:start1 :end1 :start2 :end2** *[c]* Function

These functions perform the same comparisons as **string<**, **string>**, **string<=**, and **string>=**, respectively, but without regard for character case and font. For example:

```
(string-lessp "aa" "Ab") => 1
(string-lessp "aa" "Ab" :end1 1 :end2 1) => nil
(string-not-greaterp "Aa" "Ab" :end1 1 :end2 1) => 1
```

String Construction andManipulation

8.6 The following functions are associated with the construction and manipulation of strings.

make-string *size* &key **:initial-element** *[c]* Function

This function makes a simple string of the length specified by *size* with each element initialized to the argument given to **:initial-element**, which should be a character. Although Common Lisp does not specify a default for **:initial-element**, on the Explorer system it is initialized with a code attribute of 0 (see **char-code** in paragraph 4.4, Character Construction and Attribute Retrieval).

To make character arrays that are more complex than simple strings, use **make-array**.

string-trim *char-set string* *[c]* Function
string-left-trim *char-set string* *[c]* Function
string-right-trim *char-set string* *[c]* Function

These functions return a copy of a substring of *string* with all characters in *char-set* trimmed off. With **string-trim**, the characters are trimmed off the beginning and end; with **string-left-trim**, the characters are trimmed off the beginning; and with **string-right-trim**, the characters are trimmed off the end. The *char-set* argument is a set of characters, which can be represented as a list of characters, a string of characters, or a single character. For example:

```
(string-trim #\space " Dr. No ") => "Dr. No"
(string-trim "ab" "abbafooabb") => "foo"
(string-left-trim '(#\space) " Dr. No ") => "Dr. No "
(string-left-trim "ab" "abbafooabb") => "fooabb"
(string-right-trim '(#\space) " Dr. No ") => " Dr. No"
(string-right-trim "ab" "abbafooabb") => "abbafoo"
```

Note that the order of characters in the *char-set* argument is not taken into account when characters are trimmed from the *string* argument. If any character in the beginning or end of *string* matches one of the characters in *char-set*, it is trimmed. Otherwise, a copy of *string* is returned unchanged.

Nondestructive Case Conversion Functions

8.6.1 The following functions convert the character case of a string nondestructively; that is, the original argument string is not modified, but a converted copy is returned.

string-upcase *string* &key :start :end — *[c]* Function
string-downcase *string* &key :start :end — *[c]* Function

The **string-upcase** function returns *string* with all uppercase letters, and the **string-downcase** function returns *string* with all lowercase letters. If the **:start** or **:end** argument is specified for either function, only the specified portion of the string is converted, but in any case, the entire string is returned.

If no changes are made to *string*, Common Lisp specifies that the original argument may be returned, but on the Explorer a copy of *string* is always returned. Consider the following examples:

```
(string-upcase "In the Beginning was the Word")
 => "IN THE BEGINNING WAS THE WORD"
(string-downcase "In the Beginning was the Word")
 => "in the beginning was the word"
(string-upcase "In the Beginning was the Word"
 :start 7 :end 20)
 => "In the BEGINNING WAS the Word"
```

string-capitalize *string* &key :start :end — *[c]* Function
string-capitalize *string* &key :start :end :spaces — Function

This function returns a string like *string* in which all, or the specified portion, is processed by capitalizing each word. For this function, a word is any subsequence of alphanumeric characters delimited by a nonalphanumeric character or by the end of the string. This string is capitalized by putting the first character (if it is a letter) in uppercase and any letters in the rest of the word in lowercase. If the value for :spaces is true, then hyphens are replaced with spaces and the subsequent characters are candidates for capitalization.

If no changes are made to *string*, Common Lisp specifies that the original argument may be returned, but on the Explorer a copy of *string* is always returned. Consider the following examples:

```
(string-capitalize " john") => " John"

(string-capitalize "puff the mAgiC dRAGon")
 => "Puff The Magic Dragon"

(string-capitalize 'common-lisp-zeta-lisp)
=> "Common-Lisp-Zeta-Lisp"

(string-capitalize "DON'T!")
=> "Don'T!"     ; Delimited by the quote.
```

string-capitalize-words *string* &optional *copy-p spaces* — Function

Like **string-capitalize**, this function puts each word in *string* into lowercase with an initial uppercase letter. If *spaces* is true, this function replaces each hyphen character with a space.

If *copy-p* is true (the default value), the returned value is a copy of *string*, and *string* itself is unchanged. Otherwise, *string* itself is returned, with its contents changed.

This function is somewhat obsolete. You can use **string-capitalize** followed optionally by **string-subst-char**.

Destructive Case Conversion Functions

8.6.2 The following functions convert the character case of a string destructively; that is, the original argument string is modified during conversion. Symbols are not allowed as arguments with these functions because symbol print names must not be altered.

nstring-upcase *string* &key :start :end *[c]* Function
nstring-downcase *string* &key :start :end *[c]* Function
nstring-capitalize *string* &key :start :end *[c]* Function

These functions perform the same operations as **string-upcase**, **string-downcase**, and **string-capitalize**, respectively, but alter the *string* argument.

Other String Operations

8.7 The following functions that manipulate strings are not part of the Common Lisp standard because the functionality of these operations is provided by the sequence functions described in Section 9, Sequences.

nsubstring *string start* &optional *end area* Function

This function creates an indirect array to share part of the *string* argument, beginning at *start* and going up to but not including the character specified by *end*. The default for *end* is the end of *string*. Modifying either the original string or the new substring modifies the other.

Note that **nsubstring** does not necessarily use less storage than **substring**; an **nsubstring** of any length uses at least as much storage as a **substring** that is 12 characters long. So you should not use this function only for efficiency; **nsubstring** is intended for uses in which it is important to have a substring that, if modified, causes the original string to be modified, too.

When the *area* argument is provided, it makes the array in the specified area (see paragraph 25.5, Storage and Allocation Areas).

string-append &rest *strings* Function

This function copies and concatenates any number of strings into a single string. With a single argument, **string-append** simply copies it. If there are no arguments, the returned value is an empty string. Arrays of any type can be used as arguments, and the returned value is of the same type as the first argument. Thus, **string-append** can be used to copy and concatenate any type of one-dimensional array. If the first argument is not an array (for example, if it is a character), the returned value is a string.

The corresponding Common Lisp function is **concatenate**.

substring-after-char *char string* &optional *start end area* Function

This function returns a copy of the portion of *string* that follows the next occurrence of *char* after the index specified by *start*. The copied portion ends at the index specified by *end*. If *char* is not found before *end*, a null string is returned.

The returned value is consed in *area* or in **default-cons-area** unless it is a null string. The *start* argument defaults to 0, and *end* defaults to the length of *string*.

For standard Common Lisp, see **position** and **subseq** in paragraphs 9.6, Sequence Searching, and 9.3, Elementary Sequence Functions, respectively.

string-nconc *modified-string* &rest *strings* Function

This function is like **string-append**, except that instead of making a new string containing the concatenation of its arguments, **string-nconc** modifies its first argument. The *modified-string* argument must have a fill pointer so that additional characters can be appended to it. Compare this function with **vector-push-extend**. The value returned by **string-nconc** is *modified-string* or a new, longer copy of it; in the latter case, the original copy is concatenated onto the new copy. Unlike **nconc**, **string-nconc** with more than two arguments modifies only its first argument, not every argument except the last.

In Common Lisp, use the **format** function with a *stream* argument that is a string with a fill pointer.

string-remove-fonts *fat-string* Function

This function returns a copy of *fat-string* with each character truncated to eight bits, that is, changed to font 0 with 0 control bits. If *fat-string* is of type **art-string**, nothing is changed. Typically, the *fat-string* argument is of type **art-fat-string**.

string-pluralize *string* Function

This function returns a string containing the plural of the word in the argument *string*. For example:

```
(string-pluralize "event") => "events"
(string-pluralize "Man") => "Men"
(string-pluralize "ox") => "oxen"
(string-pluralize "Can") => "Cans"
(string-pluralize "key") => "keys"
(string-pluralize "TRY") => "TRIES"
```

For words with multiple plural forms that depend on the meaning, **string-pluralize** cannot always return the proper form.

string-select-a-or-an *word* Function

This function returns `"a"` or `"an"`, depending on the string specified by *word*, whichever one appears to be correct to use before a word in English.

string-append-a-or-an *word* &rest *more-words* Function

This function returns the result of appending `"a"` or `"an"`, whichever is appropriate, to the front of the concatenation of *word* and all the *more-words*.

alphabetic-case-affects-string-comparison Variable

If this variable is true, then **string-equal**, **string-search**, and **string-reverse-search** consider case (and font) significant in comparing characters. Normally, this variable is **nil** and the string comparison functions ignore differences in case.

This variable can be bound by user programs around calls to the string comparison functions, but do not set it globally because doing so may cause system malfunctions.

String Searching

8.8 The following functions are used for string searching and character substitution.

string-search-set *char-set string* &optional *start end consider-case-p* Function

This function searches through *string* looking for a character that is in *char-set*. The *char-set* argument is a set of characters that can be represented as a list of characters, a string of characters, or a single character.

The search begins at the index *start*, which defaults to 0. The search returns the index of the first character that is **char-equal** to any element of *char-set*, or **nil** if none is found. If *end* is non-**nil**, it is assumed to be an integer and is used in place of the length of *string* to limit the extent of the search.

Case and font are significant in character comparison if *consider-case-p* is non-**nil**. In this instance, **char=** is used for the comparison rather than **char-equal**.

For standard Common Lisp, use **position-if**.

string-search-not-set *char-set string* &optional *from to consider-case-p* Function

This function is like **string-search-set** but searches for a character that is *not* in *char-set*.

For standard Common Lisp, use **position-if-not**.

string-reverse-search-set *char-set string* &optional *start end consider-case-p* Function

This function searches through *string* in reverse order for a character that is **char-equal** to any element of *char-set*. The *char-set* argument is a set of characters that can be represented as a list of characters, a string of characters, or a single character.

The search starts from an index that is one less than *start* and returns the index of the first suitable character found, or **nil** if none is found. When *start* is **nil**, the search starts at the end of *string*. Note that the index returned is from the beginning of the string, although the search starts from the end. The last (leftmost) character of *string* examined is the one at index *end*, which defaults to 0.

Case and font are significant in character comparison if *consider-case-p* is non-**nil**. In this case, **char=** is used for the comparison rather than **char-equal**.

The standard Common Lisp equivalent of **string-reverse-search-set** with a value specified for *consider-case-p* is as follows:

```
(position-if #'(lambda (x) (member x char-set :test #'char=))
             (string string-arg)
             :start start
             :end end
             :from-end (not (null start)))
```

In this example, `char-set`, `to`, and `from` correspond to the parameters of the same name in the syntax line for **string-reverse-search-set**. The `string-arg` argument corresponds to the *string* parameter in the syntax line.

For the Common Lisp equivalent of **string-reverse-search-set** with *consider-case* unspecified, use the preceding form with **char-equal** in place of **char=**.

string-reverse-search-not-set *char-set string* &optional *start end consider-case-p* — Function

This function is like **string-reverse-search-set** but searches for a character that is *not* in *char-set*.

For the Common Lisp equivalent of **string-reverse-search-not-set**, use the form specified for **string-reverse-search-set** with **position-if-not** in place of **position-if**.

string-subst-char *new-char old-char fat-string copy-p retain-font-p* — Function

This function returns a copy of *fat-string* where all occurrences of *old-char* have been replaced by *new-char*.

Case and font are ignored in comparing *old-char* with characters of *fat-string*. Normally, the font information of the replaced character is preserved, so an *old-char* in font 3 is replaced by a *new-char* in font 3. (Only **art-fat-strings** can retain a font ID other than 0.) If *retain-font-p* is **nil**, the font specified in *new-char* is stored whenever a character is replaced. The default value for *retain-font-p* is **t**.

If *copy-p* is **nil**, *fat-string* is modified destructively and returned. The default value for *copy-p* is **t**.

For standard Common Lisp, see **substitute** and **nsubstitute** in paragraph 9.5, Modifying Sequences.

String Type Functions

8.9 The following functions test whether an object is a string and coerce an object into a string.

stringp *object* — *[c]* Function

This predicate returns true if *object* is a string; otherwise, it returns **nil**. This predicate also returns true for strings of type **art-fat-string**. For example:

```
(stringp "shazam") => true
(setf gomer "shazam") => "shazam"
(stringp gomer) => true
(stringp 'gomer) => nil
(stringp "7") => true
(stringp 7) => nil
```

simple-string-p *object* — *[c]* Function

This function returns true if *object* is a string that has no fill pointers and that is not displaced. According to Common Lisp, simple arrays—and therefore simple strings—are not adjustable. However, on the Explorer system, all arrays are adjustable.

string *x* — *[c]* Function

This function coerces *x* into a string. If *x* is already a string, it is returned unchanged. If *x* is a symbol, its print name is returned. If *x* is a string-char character, then **string** returns a string consisting of that single character. If *x* is a flavor instance that accepts the **:string-for-printing** operation (such as a pathname), the result of that operation is returned.

NOTE: This function operates differently in Zetalisp mode. See Appendix A for details.

Do not use **string** when attempting to make a string from a sequence of individual characters; use **coerce** instead. (The **coerce** function does not convert symbols to strings, nor does **string** convert sequences to strings.)

Also, do not use **string** when attempting to convert an object's printed representation into a string; use **format** with a first argument of **nil** instead. (You can also use **princ-to-string** and **prin1-to-string** for this purpose.)

SEQUENCES

9

Sequence Definitions

9.1 A *sequence* is a Lisp object that contains an ordering of zero or more elements. Lists, **nil**, vectors, and therefore strings are all subtypes of type sequence. For vectors with fill pointers, the sequence is defined to be the set of active elements. Note that dotted lists are not sequences; the use of dotted lists as sequence arguments is undefined except where explicitly stated.

The functions that operate on sequences fall into two categories: those that operate on every occurrence of a single *target-item* in the sequence, and those that operate on all items that satisfy a specified predicate. As an example of the first category of sequence functions, the following form removes all occurrences of `target-item` in `some-sequence`:

```
(remove target-item some-sequence)
```

Many of the members of the second category of sequence functions are variants of members of the first category with an added **-if** or **-if-not** suffix. Rather than expecting a target item, these functions expect a *test-predicate* as a required argument (these functions do not support the **:test** and **:test-not** keywords described later). For example, the following form removes all items from `some-sequence` that are numbers:

```
(remove-if #'numberp some-sequence)
```

For functions of this second category, the test-predicate should be a function that operates with only one argument. Whenever a sequence function produces a new vector or string as its result, it produces a simple vector or simple string, respectively.

Arguments to Sequence Functions

9.2 The following information describes some general characteristics about the arguments to sequence functions:

- Optimized sequence arguments — At runtime, sequence functions must determine the type of sequence being passed as an argument. Depending on whether this argument is a list or a vector, a specialized routine is called. Although this runtime flexibility is an important feature, it is also a needless expense if you know that the arguments will always be vectors (or lists). If you can guarantee that the type of sequence will always be the same, then you can use the type specifier **the** so that the compiler can optimize the sequence call to the appropriate specialized routine. For example:

  ```
  (remove target-item (the list some-list))
  ```

- **:test, :test-not** — The argument to either of these keywords is a function that determines which elements in the sequence are to be operated on. This function should operate with only two arguments. The order of the arguments supplied to the test function is the same as the order of the arguments to the calling sequence function. In most cases, the first argument is the target item, and the second is an element from the sequence (this order is important if your test function is not commutative).

For example, the following form removes all occurrences of `bad-symbol` from the specified list:

```
(remove 'bad-symbol '(good-symbol ok-symbol bad-symbol) :test #'eq)
```

If you had used the **:test-not** keyword in this example, the result of the test condition would be logically inverted, thus removing from the list all items except `bad-symbol`. You cannot supply both the **:test** and **:test-not** keywords to the same function. If neither is specified, the default test function is **:test eql**.

- **:key** — The argument to this keyword is a function that preprocesses every element of the sequence before the test predicate is applied. The result of this function is passed as an argument to the test predicate. The function specified for **:key** should operate with only one argument. For example, the following form removes any element whose key is `bad-key` from the specified association list:

  ```
  (remove 'bad-key  '((good-key . good-datum)(bad-key . bad-datum))
         :test #'eq :key #'car)
  ```

  If the value of the **:key** argument is **nil** or unspecified, then no preprocessing is performed.

- **:from-end** — When the argument to this keyword is true, the sequence argument is conceptually processed in reverse order. That is, the result of the operation will seem to have been produced by processing the sequence in reverse order; however, this order of processing is not guaranteed. For this reason, any user-specified test functions should be free of side effects.

- **:start, :end** — The arguments to these keywords are integer indices that allow you to specify that the operation is to be performed on only a portion of the sequence argument. This portion, or subsequence, begins with the element specified by the **:start** argument and ends with the element whose index is 1 less than the **:end** argument. If the **:start** argument is unspecified or **nil**, it defaults to 0. If the **:end** argument is unspecified or **nil**, it defaults to the length of the sequence argument.

- **:start1, :end1, :start2, :end2** — The arguments to these keywords are to be used the same way as those for the **:start** and **:end** keywords but are provided for those functions that take two sequences as arguments.

- **:count** — The argument to this keyword is an integer that specifies the maximum number of items from the sequence argument that are to be processed after satisfying the test condition. (Recall that a true value is returned when the **:test** argument is satisfied, and a **nil** value is returned when the **:test-not** argument is satisfied.) Once this number has been reached, the remainder of the sequence is not tested and is therefore returned as part of the result. The default value for **:count** is **nil**, which means that every item in the sequence is to be tested.

Elementary Sequence Functions

9.3 The following functions are considered elementary operations on sequences. Most of these functions return newly constructed sequences and do not destructively modify the sequence argument. However, those functions that do destructively modify their arguments are clearly identified.

elt *sequence index* *[c]* Function

This function returns the element at the position indicated by *index* in *sequence*. The *index* argument must be a nonnegative integer smaller than the number of elements in *sequence*. Because zero-origin indexing is used, the first element in *sequence* is element number 0.

If *sequence* is a vector, **elt** observes its fill pointer if it has one. For retrieving elements with an index greater than the fill pointer, use **aref** (see paragraph 7.4, Accessing and Setting Arrays). Consider the following examples:

```
(setf sqn '((a b) c d e))
(elt sqn 0) => (a b)
(elt sqn 2) => d
```

To permanently change the value of a particular element in *sequence*, use **setf** with **elt** as follows:

```
;; Using sqn from previous example.
(setf (elt sqn 0) '(f g))
sqn => ((f g) c d e)
```

subseq *sequence start* &optional *end* *[c]* Function

This function returns a subsequence of *sequence* beginning at *start* and ending one element before *end*, if specified. Note that the element specified by *end* does *not* appear in the returned subsequence. If *sequence* is a list, the new subsequence does not share cons cells with *sequence*, so the original sequence is not destructively altered by **subseq**. The new subsequence is of the same type as *sequence*. For example:

```
(setf strng "abcdefghij")
(subseq strng 3) => "defghij"
(subseq strng 3 6) => "def"
```

You can use **setf** in conjunction with **subseq** to destructively alter a subsequence of *sequence* as follows (see also **replace** in paragraph 9.5, Modifying Sequences):

```
;;Using strng from previous example.
(setf (subseq strng 3 6) "wxyz")
strng => "abcwxyghij"
```

In this example, because the string `"wxyz"` is longer than the subsequence specified by `subseq`, this string is truncated to fit the length specified by `subseq`. Thus, `z` does not appear in `strng`. Had the substituted string been shorter than that specified by `subseq`, the balance of characters specified by `subseq` would have remained unchanged. In either case, the length of the original sequence is unchanged.

copy *object* *[c]* Function

This function generates a copy of *object*. Although **copy** returns a copy of a sequence, its implied usage is slightly more general than just copying sequences. For instance, if *object* is a structure, **copy** calls the the user-defined copy routine to generate the copy.

copy-seq *sequence* *[c]* Function

This function returns a copy of the argument *sequence*. For example:

```
(setf kong-stats '(35 118 0.268)) => (35 118 0.268)
(setf imposter-stats (copy-seq kong-stats)) => (35 118 0.268)
(eq imposter-stats kong-stats) => nil
```

length *sequence* *[c]* Function

This function returns the length of *sequence* as an integer. For a vector with a fill pointer, this is the fill pointer value. Note that for lists, the number of elements is defined to be the number of cons cells; thus, the last item in a dotted list is not considered an element. For example:

```
(length '(a b c)) => 3
(length '(a b . c)) => 2
(length "abc defg") => 8
(length '("abc" "def")) => 2
```

make-sequence *type size* &key **:initial-element** *[c]* Function

This function returns a sequence of the type specified by *type* (*type* must be a type specifier of some kind of vector or list) containing the number of elements specified by *size*. If it is supplied, the **:initial-element** argument specifies the initial value for each element in the new sequence and must be a valid object for the type of sequence indicated by *type*. For example:

```
(setf player-game-stats (make-sequence '(vector integer)
                          19 :initial-element 0))
(elt player-game-stats 5) => 0
(elt player-game-stats 15) => 0
```

If the argument *type* is **list** and no **:initial-element** is provided, then the returned sequence contains **nil** for each element. If the argument *type* is some kind of vector and **:initial-element** is not specified, then the elements of the returned sequence are undefined.

Concatenating, Mapping, and Reducing Sequences

9.4 The following functions perform sequence concatenation, mapping, and reduction operations.

concatenate *result-type* &rest *sequences* *[c]* Function

This function returns a sequence that is a copied concatenation of all its sequence arguments; the order of elements in the new sequence preserves the order in which they were specified in *sequences*. The *result-type* argument indicates the type of the new sequence and must be a type of list or vector. Note that the new sequence is a *copied* concatenation, which means that if the sequence is a list, new cons cells are created for it, leaving the cons cells of the original *sequences* unchanged (unlike **append**, which uses the cons cells of its arguments to create a new sequence).

If you specify only one sequence for *sequences* whose type is already *result-type*, then this function merely returns a copy of the sequence. If you only want a type conversion when one argument is provided, then use the **coerce** function. Consider the following examples:

```
(concatenate 'list '(1 2) '#(A 3)) => (1 2 A 3)

(concatenate 'vector '(1 2) '#(A 3)) => #(1 2 A 3)
```

map *result-type function sequence* &rest *more-sequences* *[c]* Function

This function returns a sequence of type *result-type* whose elements are the result of applying *function* to successive elements in *sequence* and *more-sequences*. For example the *n*th element in the return sequence is as follows:

```
(function (elt sequence nth)
          (elt more-sequences-1 nth)
          (elt more-sequences-2 nth)
           ...)
```

The length of the return sequence is equal to the length of the shortest sequence provided as an argument to **map**.

If *function* destructively alters its arguments, it alters the elements of *sequence* and *more-sequences* one at a time, starting with element 0. Therefore, you do not have to worry about side effects occuring to elements before they are processed by *function*.

The argument *result-type*, which must be a type of list or vector, specifies the type of the returned result. If you specify *result-type* as **nil**, then *function* is executed purely for its side effects and the resulting calculations of *function* are discarded, no new sequence is produced, and **map** returns **nil**.

Compatibility Note: In earlier versions of Lisp, the function **map** did not return a value. Due to recent developments in *functional programming*, the term *map* in current literature has come to mean what in the past Lisp users have called **mapcar**. Common Lisp follows the current meaning of **map**, and what was previously called **map** is now called **mapl** in Common Lisp.

Consider the following example:

```
(setf hero-nemesis-list
      (map 'list #'list '(Mozart Holmes Batman)
                                           '(Salieri Moriarty Penguin)))
=> ((Mozart Salieri) (Holmes Moriarty) (Batman Penguin))
```

reduce *function sequence* &key **:start :end :initial-value :from-end** *[c]* Function

This function combines the elements of *sequence* using *function*, which should be a function of two arguments. First, *function* is applied to the first two elements of *sequence* to produce a result. Next, *function* is applied to this result and the third element of *sequence* to produce a second result. Then, *function* is applied to this second result and the fourth element of *sequence* to produce a third result. This procedure continues until all *sequence* elements have been processed, and then the final result is returned.

The **:start**, **:end**, and **:from-end** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions.

If **:initial-value** is specified, it acts like an extra element of *sequence*, used in addition to the actual elements of the specified part of *sequence*. It comes, in effect, at the beginning if **:from-end** is **nil**, but at the end if **:from-end** is true. In any case, the **:initial-value** element is the first element to be processed.

If there is only one element to be processed (including **:initial-value**, if supplied), that element is returned and *function* is not called.

NOTE: If there are no elements (*sequence* is of length 0 and there is no **:initial-value**), *function* is called with no arguments and its value is returned.

Consider the following examples:

```
(reduce #'+ '(1 2 3)) => 6
(reduce 'cons '(1 2 3) :from-end t) => (1 2 . 3)
(reduce 'cons '(1 2 3) :from-end t :initial-value nil) => (1 2 3)
(reduce 'cons '(1 2 3)) => ((1 . 2) . 3)
```

Modifying Sequences

9.5 The following functions are specifically designed for modifying sequences, although not all of them do so destructively. Those functions that perform destructive modification are clearly identified.

reverse *sequence* *[c]* Function

This function returns a new sequence containing the elements of *sequence* in reverse order. The new sequence is of the same type and length as *sequence*. The **reverse** function does not modify its argument, unlike the **nreverse** function, which is faster but does modify its argument. For example:

```
(reverse "foo") => "oof"
(reverse '(a b (c d) e)) => (e (c d) b a)
```

nreverse *sequence* *[c]* Function

This function destructively reverses the order of elements in *sequence*. For example:

```
(setf x "abc") => "abc"
(setf y (nreverse x)) => "cba"
(eq x y) => t
```

fill *sequence item* &key **:start :end** *[c]* Function

This function modifies the contents of *sequence* by setting all the elements to *item*. The keywords **:start** and **:end** can be specified to limit the operation to a contiguous portion of *sequence*; if this is the case, then the elements before **:start**, at **:end**, and after **:end** are unchanged. If **:end** is **nil**, the filling goes to the end of *sequence*.

The value returned by **fill** is the modified *sequence*. For example:

```
(setf l '(a b c d e ))
(fill l 'lose :start 2 ) => (a b lose lose lose)
```

replace *into-sequence-1 from-sequence-2* &key **:start1 :end1 :start2 :end2** *[c]* Function

This function destructively replaces the specified portion of *sequence-1* with a copy of the specified portion of *sequence-2*. If the specified portion of *sequence-2* is shorter than what it is to replace, the extra elements in *sequence-1* are not changed. If the specified portion of *sequence-2* is larger than what it is to replace, then the extra elements of *sequence-2* are ignored. The returned value is the modified *sequence-1*.

The **:start1**, **:start2**, **:end1**, and **:end2** keywords operate as described in paragraph 9.2, Sequence Keywords

If the two sequence arguments are the same (**eq**) sequence, then the elements to be copied are copied first into a temporary sequence (if necessary) to make sure that no element is overwritten before it is copied. The value returned by **replace** is the modified *into-sequence-1*. For example:

```
(replace '(a b c d) '(x y z) :start1 1)
=> (a x y z)

(replace '(a b c d) '(x y z) :start1 2 :end2 1)
=> (a b x d)

(setf str "Elbow")
(replace str str :start1 2 :end1 5 :start2 1 :end2 4)
=> "Ellbo"

str => "Ellbo"
```

remove *item sequence* &key **:test :test-not :start :end :count :key :from-end** *[c]* Function
delete *item sequence* &key **:test :test-not :start :end :count :key :from-end** *[c]* Function

These functions are used for eliminating elements from a sequence argument. They test the elements of *sequence* one by one, comparing them with *item*. The functions specified by the keywords **:test** and **:test-not** are used as the comparator in the argument testing. When there is a match during the comparison testing, the matching element of *sequence* is eliminated. The function **remove** copies structure as necessary to avoid modifying *sequence*, whereas **delete** can either modify the original sequence and return it or make a copy and return that. (Currently, a list is always modified, and a vector is always copied.) The **:start**, **:end**, **:count**, **:key**, and **:from-end** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions.

Do not use the **delete** function for side effects. If you want to delete `item` from `sequence`, then use the following:

```
(setf sequence (delete item sequence))
```

Consider the following examples:

```
(remove 'x '(x (a) (x) (a x)))
=> ((a) (x) (a x))

(remove 'x '((a) (x) (a x)) :key 'car)
=> ((a) (a x))
```

remove-if *predicate sequence* &key **:start :end :count :key :from-end** *[c]* Function
delete-if *predicate sequence* &key **:start :end :count :key :from-end** *[c]* Function

These functions return a sequence like *sequence* but missing any elements that satisfy *predicate*, which is a function of one argument that is applied to one element of *sequence* at a time; if *predicate* returns a true value, that element is removed. The function **remove-if** copies structure as necessary to avoid modifying *sequence*, while **delete-if** can either modify the original sequence and return it or make a copy and return that, whichever is most efficient. (Currently, a list is always modified, and a vector is always copied.)

The **:start**, **:end**, **:count**, **:key**, and **:end-from** keywords operate as described in paragraph 9.2, Arguments to SequenceFunctions.

Do not use the **delete-if** function for side effects. If you want to delete `item` from `sequence`, then use the following:

```
(setf sequence (delete-if item sequence))
```

Consider the following examples:

```
(remove-if #'plusp '(1 -2 3 -4 5 -6) :count 2)
=> (-2 -4 5 -6)

(remove-if #'plusp '(1 -2 3 -4 5 -6) :count 2 :from-end t)
=> (1 -2 -4 -6)

(remove-if #'zerop '(1 -2 3 -4 5 -6) :key #'1-)
=> (-2 3 -4 5 -6)
```

remove-if-not *predicate sequence* &key **:start :end :count** **:key :from-end** *[c]* Function
delete-if-not *predicate sequence* &key **:start :end :count** **:key :from-end** *[c]* Function

These functions are like **remove-if** and **delete-if**, except that the elements removed are those for which *predicate* returns **nil**.

Do not use the **delete-if-not** function for side effects. If you want to delete `item` from `sequence`, then use the following:

```
(setf sequence (delete-if-not item sequence))
```

remove-duplicates *sequence* &key **:test :test-not** **:start :end :key :from-end** *[c]* Function
delete-duplicates *sequence* &key **:test :test-not** **:start :end :key :from-end** *[c]* Function

The **remove-duplicates** function returns a new sequence like *sequence*, except that all but one of any set of matching elements are removed. The function **delete-duplicates** is the same as **remove-duplicates**, except that **delete-duplicates** may destructively modify and then return *sequence* itself.

Elements are compared using **:test**, a function of two arguments. Two elements match if **:test** returns a true value. Each element is compared with all the following elements and is removed if it matches any of them.

If **:start** or **:end** is used to restrict processing to a portion of *sequence*, both removal and comparison are restricted. An element is removed only if it is itself within the specified portion and matches another element within the specified portion.

If **:from-end** is true, then elements are processed (conceptually) from the end of *sequence* forward. Each element is compared with all the preceding ones and is removed if it matches any of them. For a well-behaved comparison function, the only difference **:from-end** makes is which elements of a matching set are removed. Typically, when processing begins with the start of the sequence, the last of the matching elements is kept; with **:from-end**, the first one is kept.

The **:test-not** and **:key** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions. Consider the following examples:

```
(remove-duplicates '(1 2 3 2 4 4 5)) => (1 3 2 4 5)

(remove-duplicates '((foo #\c) (bar #\$) (baz #\C))
                   :test #'char-equal :key #'second)
=> ((bar #\$) (baz #\C))
```

The **remove-duplicates** and **delete-duplicates** functions are helpful when you want to transform a sequence into a canonical form that can be used to represent a set. See paragraph 6.7.2, Lists as Sets.

substitute *newitem olditem sequence* &key **:test :test-not** **:start :end :count :key :from-end** *[c]* Function

nsubstitute *newitem olditem sequence* &key **:test :test-not** **:start :end :count :key :from-end** *[c]* Function

These functions replace *olditem* with *newitem* in the argument *sequence*. The test predicate is given *olditem* and an element from *sequence*. If this predicate returns true (or **nil** for the **:test-not** case), *olditem* is replaced with *newitem*. The function **nsubstitute** modifies the argument *sequence* while **substitute** modifies a copy of *sequence*. Note that these functions are much different from the function **replace**, which does not test arguments.

The **:start**, **:end**, **:key**, **:count**, and **:from-end** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions.

Do not use the **nsubstitute** function for side effects. If you want to substitute `newitem` for `olditem` in `sequence`, then use the following:

```
(setf sequence (nsubstitute newitem olditem sequence))
```

Consider the following examples:

```
(setf sixers-starters '(Toney Cheeks Malone Erving Barkley))
(substitute 'Threatt 'Toney sixers-starters)
=> (Threatt Cheeks Malone Erving Barkley)

(setf bullets-starters '(Johnson Malone Ruland Johnson Roundfield))

(setf bullets-starters (substitute 'Williams 'Johnson
                                   bullets-starters :count 1))
=> (Williams Malone Ruland Johnson Roundfield)
```

When making its new sequence, **substitute** copies just enough of *sequence* to avoid having to destructively modify it. For example, if all the substitutions occur in the first three elements of a 12-element sequence, then the last nine elements of both the new sequence and the original sequence share the same cons cells. Furthermore, on the Explorer if no substitutions occur, the returned sequence is **eq** to the original sequence.

For substituting in a tree structure, use **subst** and **nsubst** (paragraph 6.7.1, Substitution Within a List).

substitute-if *newitem predicate sequence* &key **:start :end** *[c]* Function
:count :key :from-end
nsubstitute-if *newitem predicate sequence* &key **:start :end** *[c]* Function
:count:key :from-end

The function **substitute-if** returns a new sequence like *sequence* but with *newitem* substituted for each element of *sequence* that satisfies *predicate*. The *sequence* argument itself is unchanged. If *sequence* is a list, only enough of it is copied to avoid changing *sequence*.

The **nsubstitute-if** function replaces elements in *sequence* itself, modifying it destructively, and returns *sequence*.

The **:start, :end, :key, :count,** and **:from-end** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions.

Do not use the **nsubstitute-if** function for side effects. If you want to substitute `newitem` for all items that satisfy `predicate` in `sequence`, then use the following:

```
(setf sequence (nsubstitute-if newitem predicate sequence))
```

Consider the following examples:

```
(substitute-if 0 #'plusp '(1 -1 2 -2 3) :from-end t :count 2)
=> (1 -1 0 -2 0)

(substitute-if 7 #'oddp '(1 2 4 1 3 4 5))
=> (7 2 4 7 7 4 7)

(substitute-if 7 #'evenp '(1 2 4 1 3 4 5) :count 1 :from-end t)
=> (1 2 4 1 3 7 5)
```

substitute-if-not *newitem predicate sequence* *[c]* Function
&key **:start :end :count :key :from-end**
nsubstitute-if-not *newitem predicate sequence* *[c]* Function
&key **:start :end :count :key :from-end**

These functions are like **substitute-if** and **nsubstitute-if,** except that the elements replaced are those for which *predicate* returns **nil.**

Do not use the **nsubstitute-if-not** function for side effects. If you want to substitute `newitem` for all items that satisfy `predicate` in `sequence`, then use the following:

```
(setf sequence (nsubstitute-if-not newitem predicate sequence))
```

Consider the following examples:

```
(substitute-if-not 7 #'oddp '(1 2 4 1 3 4 5))
=> (1 7 7 1 3 7 5)

(substitute-if-not 7 #'evenp '(1 2 4 1 3 4 5) :count 1 :from-end t)
=> (1 2 4 1 3 4 7)
```

Sequence Searching

9.6 The following functions are used for searching sequences for specific items.

find *item sequence* &key **:test :test-not :start :end :key :from-end** *[c]* Function

This function finds the first element of *sequence* that satisfies the test when compared with *item* and returns that element. The test is specified by either the **:test** or **:test-not** keyword. If no test is successful, then the function returns **nil**.

The **:start**, **:end**, **:key**, and **:from-end** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions. Consider the following examples:

```
(find 1 '(-3 -2 -1 0 1 2 3) :test #'=) => 1
(find 1 '(-3 -2 -1 0 1 2 3) :test #'> :start 3) => 0
(find 1 '(-3 -2 -1 0 1 2 3) :test #'< :start 2 :key #'1+) => 1
```

On the Explorer system, a second value is returned, which is the index of the returned element in *sequence*.

find-if *predicate sequence* &key **:start :end :key :from-end** *[c]* Function
find-if-not *predicate sequence* &key **:start :end :key :from-end** *[c]* Function

The function **find-if** finds the first element of *sequence* that satisfies the function specified by *predicate* and returns that element.

The function **find-if-not** finds the first element of *sequence* that does not satisfy the function specified by *predicate*.

The **:start**, **:end**, **:key**, and **:from-end** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions. Consider the following examples:

```
(find-if #'plusp '(-3 -2 -1 0 1 2 3 )) => 1
(find-if #'plusp '(-3 -2 -1 0 1 2 3) :from-end t) => 3
(find-if-not #'plusp '(-3 -2 -1 0 1 2 3 )) => -3
(find-if-not #'minusp '(-3 -2 -1 0 1 2 3) :from-end t) => 3
```

On the Explorer system, a second value is returned, which is the index of the returned element in *sequence*.

position *item sequence* &key **:test :test-not :start :end :key :from-end** *[c]* Function

This function is like the **find** function, but instead of returning the element of *sequence* that passes the test with *item*, it returns the index number of this element.

The **:start**, **:end**, **:key**, and **:from-end** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions. Consider the following examples:

```
(position #\A "BabA" :test #'char-equal) => 1

(position #\A "BabA" :test #'char=) => 3
```

position-if *predicate sequence* &key **:start :end :key :from-end** *[c]* Function
position-if-not *predicate sequence* &key **:start :end :key :from-end** *[c]* Function

The function **position-if** is like **find-if** but returns the index number of the first element of *sequence* that satisfies *predicate*. If the **:from-end** argument is true, then the result returned is as if the sequence were processed in reverse order. If no element is found, the function returns **nil**.

The function **position-if-not** searches for an element of *sequence* for which *predicate* returns **nil**.

The **:start**, **:end**, **:key**, and **:from-end** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions. Consider the following examples:

```
(position-if #'plusp '(-3 -2 -1 0 1 2 3)) => 4

(position-if #'plusp '(-3 -2 -1 0 1 2 3) :from-end t) => 6

(position-if-not #'plusp '(-3 -2 -1 0 1 2 3) :start 5) => nil

(position-if-not #'minusp '(-3 -2 -1 0 1 2 3) :from-end t) => 6
```

count *item sequence* &key **:test :test-not :start :end :key** *[c]* Function

This function returns the number of elements of *sequence* that match *item*. The test is specified by either the **:test** or **:test-not** keyword.

The **:start**, **:end**, and **:key** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions.

The **:from-end** keyword argument is accepted without error, but it has no effect. Consider the following example:

```
(count 4 '(1 2 3 4 5) :test #'>) => 3
```

count-if *predicate sequence* &key **:start :end :key** *[c]* Function
count-if-not *predicate sequence* &key **:start :end :key** *[c]* Function

The function **count-if** tests each element of *sequence* with *predicate* and counts how many times *predicate* returns a true value. This number is returned.

The function **count-if-not** is like **count-if** but returns the number of elements for which *predicate* returns **nil**.

The **:start**, **:end**, and **:key** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions.

The **:from-end** keyword argument is accepted without error, but it has no effect. Consider the following examples:

```
(count-if #'symbolp #(a b "foo" 3)) => 2
(count-if-not #'symbolp #(a b "foo")) => 1
```

mismatch *sequence1 sequence2* &key **:test :test-not :start1 :end1 :start2 :end2 :key :from-end** *[c]* Function

This function compares successive elements of the specified portion of *sequence1* with successive elements of the specified portion of *sequence2*, returning **nil** if they all match, or else the index in *sequence1* of the first mismatch. If the specified portions of the sequences differ in length but match for all elements compared, the value is the index in *sequence1* of the place where the shorter sequence portion ends. If the specified portion of *sequence1* is the shorter of the two, the returned value equals the length of this portion of *sequence1*, so the value returned is not the index of an actual element, but it still describes the place where comparison stopped.

If the **:test** keyword is specified, its value should be a function that operates with two arguments. This function is applied to corresponding elements in both sequences. If it returns true, the elements are considered to match, and processing continues. If the **:test-not** keyword is used, then the function must return **nil** for processing to continue; in other words, the function stops when the first match is found in the sequences. For example:

```
(mismatch '(1 2 3 4 5) '(5 4 3 2 1) :test-not #'=) => 2
```

The **:start1**, **:start2**, **:end1**, and **:end2** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions.

If the **:key** keyword is specified, its value should be a function that operates with one argument. This function is applied to each element of both sequences. The value returned from this function is passed on as an argument to the **:test** function.

If the **:from-end** argument is true, the comparison proceeds conceptually from the end of each sequence or portion. The first comparison uses the last element of each sequence portion; the second comparison uses the next-to-the-last element of each sequence portion, and so on. When a mismatch is encountered, the value returned is *one greater than* the index of the first mismatch encountered in order of processing (closest to the ends of the sequences).

Consider the following examples:

```
(mismatch "Foo" "Fox") => 2
(mismatch "Foo" "FOO" :key #'char-upcase) => nil
(mismatch '(a b) #(a b c)) => 2
(mismatch "Win" "The Winner" :start2 4 :end2 7) => nil
(mismatch "123..123" "123" :from-end nil) => 3
(mismatch "123..123" "123" :from-end t) => 5
```

search *for-sequence-1 in-sequence-2* &key **:from-end :test :test-not :key :start1 :end1 :start2 :end2** *[c]* Function

This function searches *in-sequence-2* (or a portion of it) element by element for a subsequence that matches *for-sequence-1*. The value returned by **search** is the index in *in-sequence-2* of the beginning of the matching subsequence. If no matching subsequence is found, the returned value is **nil**. The comparison of each subsequence of *in-sequence-2* is made with **mismatch**, and the **:test**, **:test-not**, and **:key** arguments are used only to pass parameters to **mismatch**.

Normally, subsequences are considered to start with the beginning of the specified portion of *in-sequence-2* and to proceed toward the end. The value is therefore the index of the earliest subsequence that matches. If **:from-end** is true, the subsequences are processed in the reverse order, and the value returned identifies the last subsequence that matches. In either case, the value identifies the beginning of the subsequence found.

The **:start**, **:end**, and **:key** keywords operate as described in paragraph 9.2, Arguments to Sequence Functions.

Consider the following example:

```
(search '(#\A #\B) "cabbage" :test #'char-equal) => 1
```

Sorting and Merging

9.7 The following functions are provided for sorting vectors and lists. These functions use algorithms that always terminate no matter which sorting predicate is used, provided that the predicate always terminates.

The main sorting functions are not *stable*; that is, equal items may have their original order changed. If you want a stable sort, use the stable versions of these functions; however, note that stable algorithms are slower.

After sorting is completed, the argument (be it a list or a vector) is rearranged internally so that it is completely ordered. Vectors are ordered by permutation of the elements; lists are ordered by use of **rplacd**. When using these methods of sorting, ensure that you sort a copy of the sequence argument (obtained by using **copy-seq**), unless you want to destructively modify the original sequence. Furthermore, **sort** invoked on a list should not be used for side effects; the result is conceptually the same as the argument but in fact is a different Lisp object.

If you supply a predicate that destructively alters the sequence and this predicate produces an error during execution, you cannot recover the original sequence unless you can correct the problem from within the error handler.

The sorting package can process cdr-coded lists and sorts them as if they were vectors.

sort *sequence predicate* &key :key *[c]* Function

This function reorders the elements of *sequence*, according to the function specified by *predicate*, and returns a modified sequence. The *predicate* argument must be applicable to all the objects in the sequence. This predicate should accept two arguments and should return a true value only if its first argument is less (in some appropriate sense) than its second. If **:key** is specified, it should be a function of one argument. Each element in *sequence* is passed to this function and the returned value is passed to the *predicate* function.

The following example sorts a list alphabetically by the first atom found at any level in each element:

```
(defun get-symb (x)
   (if (symbolp x)
        x
        (get-symb (car x))))

(sort all-stars
       #'(lambda (x y)
              (string-lessp (get-symb x) (get-symb y))))
```

Suppose `all-stars` contains these elements before the sort:

```
((Julius Erving) (Philadelphia 76ers))
((Moses Malone) (Philadelphia 76ers))
((Larry Bird) (Boston Celtics))
((Sidney Moncrief) (Milwaukee Bucks))
((Isiah Thomas) (Detroit Pistons))
((Dominique Wilkins) (Atlanta Hawks))
```

Then, after the sort, `all-stars` contains the following:

```
((Dominique Wilkins) (Atlanta Hawks))
((Isiah Thomas) (Detroit Pistons))
((Julius Erving) (Philadelphia 76ers))
((Larry Bird) (Boston Celtics))
((Moses Malone) (Philadelphia 76ers))
((Sidney Moncrief) (Milwaukee Bucks))
```

When **sort** is given a list, it may change the order of the conses of the list (using **rplacd**), so it cannot be used merely for side effects; only the *returned value* of **sort** is the sorted list. Because cons cells are modified, a symbol bound to the original list may have some of its elements missing when **sort** returns. If you need both the original list and the sorted list, you must copy the original and sort the copy (see **copy-list** in paragraph 6.4, Functions Associated With Lists).

If the *sequence* argument is a vector with a fill pointer, note that, like most sequence functions, **sort** considers the active length of the vector to be the length, so only the active part of the vector is sorted.

sortcar *sequence predicate* Function

This function is the same as **sort**, except that the predicate is applied to the cars of the elements of *sequence* instead of directly to the elements of *sequence*. For example:

```
(sortcar '((3 . dog) (1 . cat) (2 . bird)) #'<)
=> ((1 . cat) (2 . bird) (3 . dog))
```

Remember that **sortcar**, when given a list, may change the order of the conses of the list (using **rplacd**), so it cannot be used merely for side effects; only the *returned value* of **sortcar** is the sorted list. The original list is destroyed by sorting.

A portable Common Lisp program should use the following equivalent form:

`(sortcar` *sequence predicate*`) <=> (sort` *sequence predicate* `:key #'car)`

stable-sort *sequence predicate* *[c]* Function

This function is like **sort**, but if two elements of *sequence* are equal (*predicate* returns **nil** when applied to them in either order), then they remain in their original order.

stable-sortcar *sequence predicate* Function

This function is like **sortcar**, but if two elements of *sequence* are equal (*predicate* returns **nil** when applied to their cars in either order), then they remain in their original order.

A portable Common Lisp program should use the following equivalent form:

```
(stable-sortcar sequence predicate) <=>
(stable-sort sequence predicate :key #'car)
```

merge *result-type sequence1 sequence2 predicate* &key **:key** *[c]* Function

This function returns a single sequence containing the elements of *sequence1* and *sequence2* interleaved in order according to *predicate*. The length of the result sequence is the sum of the lengths of *sequence1* and *sequence2*. The *result-type* argument specifies the type of sequence to create, as in **make-sequence**.

The interleaving is performed by inserting into the returned sequence the next element of *sequence1* unless the next element of *sequence2* is *less than* the element of *sequence1* according to *predicate*. Therefore, if each of the argument sequences is sorted, the result of **merge** is also sorted.

The **:key** keyword operates as described in paragraph 9.2, Arguments to Sequence Functions. Consider the following example:

```
(merge 'string "Abd" "Cef" #'char< :key #'char-upcase)
=> "AbCdef"
```

The following two functions do not work on general sequences. They are documented here to provide the complete set of sorting functions.

sort-grouped-array *array group-size predicate* Function

This function considers its array argument to be composed of records of *group-size* elements each. These records are considered as units and are sorted with respect to one another. The *predicate* is applied to the first element of each record, so the first elements act as the keys on which the records are sorted.

sort-grouped-array-group-key *array group-size predicate* Function

This function is like **sort-grouped-array**, except that the *predicate* is applied to four arguments: an array, an index into that array, a second array, and an index into the second array. The *predicate* function should consider each index as the subscript of the first element of a record in the corresponding array and should compare the two records. This function is more general than **sort-grouped-array** because the function can access all of the elements of the relevant records instead of only the first element.

Sequence Predicates

9.8 The following functions are similar to mapping functions in that some or all of a sequence's elements are tested with a specified predicate. These functions differ in that they are used as predicates, not as mapping functions.

every *predicate sequence* &rest *more-sequences* *[c]* Function

This function returns **nil** for the first element of *sequence* that fails the test specified by *predicate*. If every element of *sequence* passes the test, then **every** returns a true value. If *more-sequences* are specified, **every** uses *predicate* to test the first elements of all the sequences, then all the second elements, and so on until some element fails the test or until the shortest sequence is exhausted. The test specified for *predicate* must accept the same number of arguments as there are sequences specified for **every**. For example:

```
(every 'plusp '(-4 0 5 6)) => nil
(every 'plusp '(5 6)) => true
```

In Zetalisp mode, **every** has a somewhat different meaning; refer to Appendix A for details.

some *predicate sequence* &rest *more-sequences* *[c]* Function

This function returns true if any element of *sequence* passes the test specified by *predicate*. If *more-sequences* are specified, **some** uses *predicate* to test the first elements of all the sequences, then all the second elements, and so on until some element passes the test or until the shortest sequence is exhausted. The test specified for *predicate* must accept the same number of arguments as there are sequences specified for **some**. For example:

```
(some 'plusp '(-4 0 5 6)) => true
(some '> '(-4 0 5 6) '(0 12 12 12)) => nil
(some '> '(-4 0 5 6) '(3 3 3 3)) => true
(some '> '(-4 0 5 6) '(3 3)) => nil
```

In Zetalisp mode, **some** has a somewhat different meaning; refer to Appendix A for details.

notany *predicate sequence* &rest *more-sequences* *[c]* Function

This function returns **nil** for the first element of *sequence* that passes the test specified by *predicate*. If none of the elements of *sequence* pass the test, **notany** returns a true value. If *more-sequences* are specified, **notany** uses *predicate* to test the first elements of all the sequences, then all the second elements, and so on until some element passes the test or until the shortest sequence is exhausted. The test specified for *predicate* must accept the same number of arguments as there are sequences specified for **notany**.

notevery *predicate sequence* &rest *more-sequences* *[c]* Function

This function returns a true value for the first element of *sequence* that fails the test specified by *predicate*. If all the elements of *sequence* pass the test, then **nil** is returned. If *more-sequences* are specified, **notevery** uses *predicate* to test the first elements of all the sequences, then all the second elements, and so on until some element fails the test or until the shortest sequence is exhausted. The test specified for *predicate* must accept the same number of arguments as there are sequences specified for **notevery**.

STRUCTURES

10

Introduction

10.1 The data type **structure** defines a data-organizing object within the Lisp environment. The prime benefit of structures is that they allow data structures to be referenced as abstract objects. Specifically, the implementation details of the data structure are hidden from the user, while the designer of the structure still controls the key aspects of storage allocation and naming conventions for the various support routines.

Note that structures are generally implemented as sequences and can be manipulated by the generic sequence operations. Additionally, when you know exactly what kind of sequence (lists or vectors) is being used, you can also use support functions for the appropriate data types. Generally, this practice should be avoided in the interest of data abstraction.

The **defstruct** macro, which defines a structure, requires an unusually large amount of documentation because it has so many options and support features. However, in its simplest form, it is quite easy to use and remember.

The defstruct Macro

10.2 The explanation of **defstruct** is divided into three parts. The following definition explains the calling sequence to the macro itself. After this definition is a description of each support feature, which are an important part of the **defstruct** facility. Finally, the **defstruct** options and support functions are explained.

defstruct *name* [*doc-string*] {*slot-description*}* *[c]* Macro
defstruct (*name* {*option value*}*) [*doc-string*] {*slot-description*}* *[c]* Macro

This macro defines a structure according to the specified arguments. None of these arguments are evaluated. The *name* argument, which must be a symbol, specifies the name of the structure. If no options are specified, then the name argument can appear by itself. Otherwise, the first argument to **defstruct** is a list whose first element is *name* and whose remaining elements are option specifications. These options are discussed in paragraph 10.4, **defstruct** Options.

The *doc-string* argument, if supplied, is associated with the structure definition and can be accessed with this form:

```
(documentation 'name 'structure)
```

The *doc-string* argument can be updated with this form:

```
(setf (documentation 'name 'structure) "new documentation")
```

The *slot-description* argument can be any of the following forms:

- *slot-name*
- (*slot-name* [*default-init-form* {*slot-option*}*])
- ({(*slot-name byte-spec* [*default-init-form* {*slot-option*}*])}*)

For each of these forms, *slot-name* must be a symbol, and all of the *slot-names* must be unique for a given structure definition. When structures of this type are created, they can supply an initial value for each slot. If no initial value is supplied to the constructor function, then the *default-init-form* is evaluated and its returned value is used as the initial value for the slot. Besides supplying an initial value with the constructor form or using a *default-init-form*, there are several other ways of using a **defstruct** options to establish an initial value for a slot. If no initial value is made available for a slot, Common Lisp states that the initial value is undefined. On the Explorer system, slots without a *default-init-form* are initialized to an appropriate value depending on the type of the structure. For more details on the implementation type, see the explanation of the **:type defstruct** option in paragraph 10.4, **defstruct** Options.

The third form for specifying a *slot-description* listed previously allows you to define explicitly how slots can be stored in memory using byte specifiers, which are also called *byte-specs*. This feature is an Explorer extension and is not part of the Common Lisp standard. The use of byte specs is explained in paragraph 10.5, Byte Fields.

The *slot-options* are a series of alternating keyword names and their associated values.

- **:type** — The corresponding value of this keyword must be a valid type specifier that declares what the slot's data type will be. On the Explorer system, this type restriction is not enforced, although the information is sometimes used to select storage allocation schemes or code that references the slot.

- **:read-only** — If the corresponding value of this keyword is true, then the associated slot value cannot be updated once the structure is created. More specifically, the accessor form for this slot is not **setf**able.

- **:documentation** — The corresponding value of this keyword should be a documentation string for the associated slot access function, which is described later. This option is an Explorer extension and is not part of the Common Lisp standard.

Consider the following examples:

```
(defstruct sailboat beam length-over-all sail-area)

(defstruct yacht
   (beam 10)
   (length-over-all 34 :type (integer 34)))
```

The definition of `sailboat` has three slot values: `beam`, `length-over-all`, and `sail-area`. The `yacht` definition has two slots: `beam`, which defaults to 10, and `length-over-all`, which defaults to 34 and which should always be a positive integer not less than 34.

defstruct Features

10.3 Each of the following **defstruct** features is automatically made available as a result of evaluating a **defstruct** form. For the sake of discussion, consider the following **defstruct** form:

```
(defstruct yacht
   (beam 10)
   (length-over-all 34 :type (integer 34)))
```

The Constructor

10.3.1 This form defines a structure-creating function whose default name is **make-** followed by the structure name; in this case, it is `make-yacht`. To initialize any of the slot values, you supply as arguments the slot name (in keyword format) followed by the desired initial value. Thus, the following examples create data structures of type `yacht`:

```
(setf nice-boat (make-yacht))      ; Take the defaults.

(setf courageous (make-yacht :beam 20
                             :length-over-all 90))
```

Data Type

10.3.2 When the **defstruct** is defined, the data type `yacht` is defined, allowing you to use `yacht` as the *type* argument to **typep**. Thus, if COURAGEOUS is a symbol whose value is an object of type `yacht`, you could test for this type with the following form:

```
(typep courageous 'yacht) => true
```

Type Predicate

10.3.3 The **defstruct** macro also defines its own predicate function whose default name is the name of the structure with a **-p** suffix. Thus, the previous example could be written as follows:

```
(yacht-p courageous) => true
```

Accessor Functions

10.3.4 For each slot name in the structure, an accessor function is defined. The name of the accessor function is a concatenation of the structure name and slot name joined by a hyphen. These accessor functions accept one argument, which should be an object of the indicated structure type. Additionally, these accessor functions can be used as *place* arguments to **setf** in order to update the slot values. For example:

```
(yacht-beam courageous) => 20
(setf (yacht-beam courageous) 11) => 11
```

Copy Function

10.3.5 In addition to the constructor function, a structure copy function is defined whose default name is **copy-** followed by the name of the structure. This copy function expects one argument, which should be an object of the indicated structure type. A call to this function creates a copy of the structure, but the values of the copy's slots are **eql** to the corresponding slots in the original structure. The following form shows how to copy a structure:

```
(setf aus-2 (copy-yacht courageous))
```

#S Reader Macro

10.3.6 Besides the **defstruct** macro, the Lisp Reader #S also creates structures. The syntax is as follows:

#S(*structure-type-name* {*slot-name-keyword value*}*)

In this form, *structure-type-name* is a defined structure, *slot-name-keyword* is the name of a slot in that structure (with a colon prefix), and *value* is an acceptable value for that slot. By default, the **print** function uses this same format to display a structure object. Frequently, not all slot names are present in the printed representation because the current value is **eql** to the *default-init-form*. Thus, if the form is read back in, it will be correctly constructed. However, if you change your *default-init-form* in the **defstruct** and read the form back in, you get the new default value.

```
;;; Specify all arguments.
#S(yacht :beam 12 :length-over-all 42)
=> #S(yacht :beam 12 :length-over-all 42)

;;; Accept the default for all arguments.
#S(yacht) => #S(yacht)

;;; Note that the explicitly supplied value is not printed because it
;;; matches the default.
#S(yacht :beam 10) => #S(yacht)
```

print-structure Variable

When the value of this variable is true (the default), structure objects are printed in the following format:

#S(*structure-name slot-name slot-value* ...)

When the value of this variable is **nil**, structure objects are printed as follows:

#<object-name address>

To provide compatibility with other Lisp dialects, structures are printed according to the value of ***print-array*** if ***print-structure*** is unbound.

defstruct Options

10.4 The options to **defstruct** are supplied in a list as the first argument. The first element of this list is the name of the structure. The remaining elements are either keyword options or a sublist whose first element is a keyword option and whose remaining elements are its arguments. Thus, for a structure using options, the syntax for the first argument to **defstruct** is as follows:

(*name* {*keyword-option*|(*keyword-option* {*arg*}*)}*)

In this form, *name* is the name of the structure being created and the *keyword-options* are any of those listed in paragraphs 10.4.1, Common Lisp **defstruct** Options and 10.4.2, Explorer Extension **defstruct** Options. Recall that none of the arguments are evaluated in the **defstruct** macro expansion.

Common Lisp defstruct Options

10.4.1 The following options to the **defstruct** macro are part of the Common Lisp standard.

- **:conc-name** *[c]* — This option is used to supply the name that is concatenated to the beginning of the accessor functions. If unsupplied, this option defaults to the name of the structure. This option accepts one argument, which should be a string or a symbol. Note that if you want a hyphen between the **:conc-name** value and the slot name, the **:conc-name** argument must include this hyphen. If you supply a value of **nil** for **:conc-name**, then the empty string ("") is implied. In this case, each of the slot names becomes the name of its accessor function, so you should choose slot names that do not conflict with existing functions. If you supply a string, you should probably use all uppercase letters; if you use lowercase letters, then the symbols that are created for the accessor functions also have lowercase letters, which is a problem because the Reader normally maps all lowercase letters to uppercase.

- **:constructor** *[c]* — This option allows you to name the constructor function for the structure. An extra option also allows you to change the calling sequence of the constructor to a position-dependent scheme rather than keyword assignment. If the constructor option is not supplied, the constructor name is a concatenation of **make-** and the structure name. If the argument to this option is **nil**, then no constructor function is defined. This option has the following syntax:

  (**:constructor** [*constr-name* [*lambda-list*] [*doc-string*]])

  In this form, *constr-name* should be a symbol (on the Explorer system, it can also be a function spec), and *doc-string* is the documentation string to be associated with the constructor function. If the *lambda-list* argument is not supplied, then new instances of the structure are created by calling **make-*structure-name*** (or *constr-name* if specified) with alternating slot names (in keyword form) and the values to which they should be set. For example:

```
(defstruct (yacht (:constructor 'buy-a-boat))
     beam length-over-all)

(buy-a-boat :beam 20 :length-over-all 90)
```

  You can supply more than one constructor option with **defstruct**, thus allowing you to have more than one creation function each with its own name and argument list. It is important to have at least one constructor that uses keywords (that is, no lambda list) so that the **#S** Reader macro can create instances of this structure.

  If the *lambda-list* argument is specified, then the constructor function expects its arguments in a position-dependent order rather than by keyword assignment. The *lambda-list* becomes the lambda list for the constructor function. You are allowed to use &optional, &rest, and &aux lambda-list keywords. Each of the parameters or auxiliary variables declared in the *lambda-list* should correspond to a slot name. You are not allowed to specify *supplied-p* arguments. If you specify a lambda list, then the **:make-array** and **:times** arguments to the **:constructor** option cannot be specified (see the **:make-array** and **:times** options in paragraph 10.4.2, Explorer Extension **defstruct** Options). The following list specifies the precedence of the ways to initialize a slot:

  1. Explicitly supplied arguments to the constructor function.
  2. Initialization in the constructor lambda list. (This applies only to &optional and &aux parameters.)
  3. Initialization in **:include** option overrides for slots defined in substructures. (Note that the constructor lambda list for an included structure is not considered.)
  4. Init-forms in the slot descriptor including those in substructures.
  5. Use of the default initial value for the structure type, which is implementation dependent.

Consider the following example:

```
(defstruct (yacht (:constructor buy-a-boat
                      (price &optional
                             (length-over-all (/ price 1000))
                             beam
                             &rest other-properties
                             &aux (year-model 1987))
                      "Doc string for the buy-a-boat function."))
 price (beam 10) (length-over-all 34) other-properties year-model))

(buy-a-boat 20000)         ; Buy a boat for $20,000.
=> #S(yacht :price 20000 :length-over-all 20
            :beam 10 :other-properties nil :year-model 1987)
```

In this example, five slots are initialized. The `price` is explicitly supplied, `length-over-all` is calculated as a function of `price` by the constructor init-form, `beam` is initialized with the slot init-form, `other-properties` defaults to `nil`, and `year-model` is set to `1987`. Note that in this constructor, `year-model` is not settable by the caller. Using this scheme, you can perform operations such as recording the creation date of the structure by setting a slot variable to the returned value of **get-universal-time**.

Common Lisp refers to this form of constructor as a *By Ordered Argument* constructor, or BOA constructor.

- **:copier** *[c]* — This option allows you to name the copier function for the structure. If this option is unsupplied, the copier name is a concatenation of **copy-** and the name of the structure. The option can accept one argument, which should be a symbol (on the Explorer system, a string is also allowed). If the argument to this option is **nil**, no copier function is defined.

  If the copier function is created, it takes one argument, which should be a structure of the appropriate type. The returned value is another structure of the same type whose slot values are **eql** to those of the original structure.

- **:predicate** *[c]* — This option allows you to name the predicate function for the structure. If this option is not supplied, the predicate name is a concatenation of the structure name with a **-p** suffix. This option accepts one argument, which should be a symbol (on the Explorer a string is allowed). If the argument to this option is **nil**, no predicate function is defined.

  If the predicate function is created, it accepts one argument, which can be any object. The returned value is true if the object is a structure of the appropriate type. Note that the use of this option does not affect the data type name, which is always the name of the structure.

- **:include** *[c]* — This option allows a structure definition to include slot definitions from another structure, which is called a *substructure*. Only one include statement can appear in each **defstruct** options list. The syntax for the **:include** option is as follows:

  (**:include** *substructure-name* {*slot-description*}*)

If present, the **:include** option must be given at least one argument, which should be a structure type. An object of the type being defined combines the locally declared slots with those of the substructure in a transparent way. For instance, the accessors for slot names defined in the substructure are a concatenation of the new structure's **:conc-name** value and the slot name of the substructure. Thus, the slot names of the included structure must be different from the slot names in the including structure. The type being defined is considered to be a subtype of the included structure type. Consequently, an object of this new type can be used as an argument to the accessor functions of the substructure. Consider the following examples:

```
(defstruct boat (beam 10) length-over-all)

(defstruct (sailboat (:include boat))
           sail-area)

(subtypep 'sailboat 'boat) => true

(setf my-boat (make-sailboat
                 :length-over-all 34 :sail-area 300))
(sailboat-beam my-boat) => 10
(boat-beam my-boat) => 10
```

It is possible to override the default values and slot options of the substructure. To do so, simply include a slot descriptor after the name of the substructure in the **:include** argument list. For example:

```
(defstruct (sailboat (:include boat (beam 15))))

(sailboat-beam (make-sailboat)) => 15
```

The following rules for **:include** slot descriptor arguments must be observed:

- The symbol used for the slot name must be defined in the substructure. The symbol can be a keyword, in which case the symbol name of the keyword is expected to be a slot name in the substructure.
- A slot that is read-only in the substructure must also be read-only in the overriding slot description.
- If a type is specified in the overriding slot descriptor, it must be the same type or a subtype of what is allowed in the substructure.

Otherwise, the overriding is intuitive. For instance, if the overriding form does not include a *default-init-form*, the new slot will not have one even if the substructure does have a *default-init-form*. Another possibility is to make the overriding slot definition read-only instead of updatable as the substructure slot is.

- **:print-function** *[c]* — This option allows the user to specify a function to print a structure of this type. The print function should take three arguments: the structure to be printed, the stream on which to print it, and the current printing depth (which should be compared with ***print-level*** and ***print-structure*** to decide when to stop recursing). The function is also expected to observe the values of the various printer-control variables, such as ***print-escape*** and ***print-pretty***.

This option cannot be used if the **:type** option is used. However, if neither **:print-function** nor **:type** is used, the default print function prints the structure in the #S format (see paragraph 10.3, **defstruct** Features).

- **:type** *[c]* — This option allows the user to specify the representation of the structure. Specifically, the slots are stored in the order in which they are defined using the prescribed sequence implementation. If you use the **:type** option, the structure name is not remembered and does not become a valid type specifier, nor is a predicate function defined unless you also use the **:named** option.

  A typical use of this option is for generating a mapping of an existing data structure into a **defstruct** definition so that the various support functions can be used with the existing data structure. If this is your intention, then you should use the **:type** option because Common Lisp states that otherwise the representation is implementation dependent. Thus, the mapping to your data structure might not be portable to other Common Lisp implementations. Moreover, when you use this option, arguments to accessor functions are not type checked (except to verify that the sequence is long enough).

  This option takes one argument, which, in Common Lisp, should be one of the following type specifications:

  - **vector** — This type specifier causes the structure to be implemented using a general vector.

  - (**vector** *element-type*) — This type specifier causes the structure to be implemented as a vector that can be optimized to hold elements of type *element-type*.

  - **list** — This type specifier causes the structure to be implemented as a list.

  The Explorer system supports several other type specifications that are discussed in paragraph 10.4.2, Explorer Extension **defstruct** Options. Notice that the preceding Common Lisp specifications are not prefixed with colons as the extension specifications are.

  If the structure does not use the **:named**, **:include**, or **:initial-offset** option, then the slots are stored starting in the first element of the sequence. If a substructure is included, then all of the space for the substructure is allocated first, including its name and initial offset, if supplied.

- **:named** *[c]* — This option specifies that the structure name is to be stored in the structure. This option is used by default unless you select the **:type** option. The **:named** option takes no arguments.

  Note that storing the structure name is different from declaring the structure name as a valid type specifier. This is the functional difference between the Common Lisp named structure and the various named structure types discussed later as Explorer extensions to the **:type** option.

  If a structure uses the **:named** option and the **:type** option with a Common Lisp defined argument, then the structure name is stored in the first element of the sequence, the first slot is stored in the second element of the sequence, the second slot is stored in the third element of the sequence, and so on. If you also use **:initial-offset**, then the offset is allocated starting from the first element of the sequence, followed by the structure name and then the slots. Note that this method of storage implies that the argument to **:type** is either **list** or a vector whose *element-type* allows a symbol (the structure name) to be stored as an element.

- **:initial-offset** *[c]* — This option tells **defstruct** to skip over a certain number of elements in the storage representation before it allocates the first slot in the structure. This option requires one argument, which must be a nonnegative integer, to indicate the number of elements to be skipped. This option can be used only if the **:type** option is also used.

Explorer Extension defstruct Options

10.4.2 The following are the options to the **defstruct** macro that are Explorer extensions and are not part of the Common Lisp standard.

- **:type** — Although this option is provided for in Common Lisp, the Explorer system supports the following additional structure types:
    - **:named-array**
      :array — These type specifiers are like the Common Lisp vector argument with and without the **:named** option, respectively. However, **:named-array** has the following differences:
        - The structure name is remembered as a legal data type.
        - If **:named-array** is used with **:initial-offset**, the name is allocated before the offset.
    - **:named-typed-array**
      :named-vector
      :typed-array
      :vector — The first two of these are the same as the Common Lisp **vector** argument with the **:named** option, and the last two are the same as the Common Lisp **vector** argument without the **:named** option. When the structure is named, however, it differs as follows:
        - The structure name is remembered as a legal data type.
        - The name is stored in element 1 of an array leader that is associated with the array that holds the slots.
    - **:named-list**
      :list — These are the same as the Common Lisp **list** argument with and without the **:named** option, respectively. If this structure type is named and if **:initial-offset** is used, then the name is allocated before the offset.
    - **:list*** — This is the same as the Common Lisp **list** option except that the last slot of the structure is stored in the cdr of the last cons cell, thus making the structure a dotted list (unless, of course, the last slot contains a list).
    - **:named-array-leader**
      :array-leader — These types are like **:named-typed-array** and **:typed-array** except that the data is stored in the array leader, with the first slot (or initial offset, if specified) stored in element 0 of the

leader and so on. The type and size of the associated array is left to the user's discretion. See the **:make-array** option discussed later. The **:named-array-leader** option has the following differences:

  - The structure name is remembered as a legal data type.
  - If the array is named, then the name is stored in element 1 of the leader, but element 0 is still used to store the first slot (or initial offset), and the second element (or offset) is stored in element 2.

- **:named-fixnum-array**
  :fixnum-array — These types are like the Common Lisp **vector** argument, but the type of the vector is **art-fix**. The **:named-fixnum-array** has the following differences:
  - The structure name is remembered as a legal data type.
  - If **:named-fixnum-array** is used with **:initial-offset**, then the name is allocated before the offset.

- **:named-flonum-array**
  :flonum-array — These types are like the Common Lisp **vector** argument, but the type of the vector is **art-single-float**. The **:named-flonum-array** option has the following differences:
  - The structure name is remembered as a legal data type.
  - If **:named-flonum-array** is used with **:initial-offset**, then the name is allocated before the offset.

- **:tree** — This type specifies that the structure is implemented as a binary tree of cons cells where each leaf holds a slot.

- **:fixnum** — This unusual type implements the structure as a single fixnum. The structure can have only one slot. This type is useful only with the byte-field feature (discussed in paragraph 10.5, Byte Fields); it lets you store several numbers within fields of a fixnum by specifying the field names.

- **:grouped-array** — This type allows you to store several instances of a structure side-by-side within an array. However, this feature is somewhat limited: it does not support the **:include** and **:named** options.

  The accessor functions are designed to take an extra argument, which should be an integer and which is the index indicating where in the array this instance of the structure starts. This index should normally be a multiple of the size of the structure. Note that the index is the *first* argument to the accessor function and that the structure is the *second* argument. This order is used because the structure is &optional if the **:default-pointer** option is used.

  Note that the *size* of the structure (for purposes of the **:size-macro** option) is the number of elements in *one* instance of the structure. The actual length of the array is the product of the size of the structure and the number of instances. The number of instances to be created by the constructor is taken from the **:times** keyword of the constructor or the argument to the **:times** option to **defstruct**.

  - :phony-named-vector — This type is the same as the Common Lisp **vector** argument with the **:named** option.
  - (*type subtype*) — This type is equivalent to specifying *type* as the argument to **:type** and *subtype* as the argument to **:subtype**. For example, `(:type (:array (mod 16.)))` specifies an array of four-bit bytes.

- **:times** — This option is used for structures of type **:grouped-array** to control the number of instances of the structure to be allocated by the constructor (see the previous description of **:grouped-array**). Noncallable constructors (macros) also accept a keyword argument **:times** to override the value given in the **defstruct**. If **:times** appears in neither the invocation of the constructor nor as a **defstruct** option, the constructor allocates only one instance of the structure.

- **:subtype** — For structures that are arrays, **:subtype** allows you to specify the array type. This option requires one argument, which must be either an array type name, such as **art-4b**, or a type specifier restricting the type of elements stored in the array. In other words, it should be a suitable value for either the *type* or the *element-type* argument to **make-array**.

  If no **:subtype** option is specified but a **:type** slot option is given for every slot, **defstruct** may deduce a subtype automatically to make the structure more compact.

- **:alterant** — This option defines an alterant macro for the structure, allowing you to write code that modifies several fields in a structure at once. If this option is not specified or is supplied as an option without arguments, then the name of the alterant macro is a concatenation of **alter-** and the structure name. Note that this provides an alterant macro even if you have otherwise strictly conformed to a Common Lisp calling sequence. This option accepts one argument, which can be a symbol or a string. If the argument is the symbol **nil**, then no alterant macro is created.

  The benefits of the alterant macro are that it looks cleaner than using multiple setfs and that it can sometimes be expanded into more efficient code. The syntax for the alterant macro is as follows:

  (**alter-***structure-name structure-object* {*slot-name-keyword form*}+)

  Thus, to alter a `yacht` structure, you would use the following form:

```
(alter-yacht COURAGEOUS :beam 30 :length-over-all 150)
```

- **:default-pointer** — Normally, the accessors defined by **defstruct** expect to be given exactly one argument. However, if you use the **:default-pointer** argument, the argument to each accessor is optional. If the accessor is used with no argument, it evaluates the default-pointer form to find a structure and then accesses the appropriate slot of that structure. For example:

```
(defstruct (player
             (:default-pointer *default-player*))
   name
   position)

(macroexpand '(player-name x)) ==> (aref x 0)
(macroexpand '(player-position)) ==> (aref *default-player* 1)
```

  If no argument is given to **:default-pointer**, the name of the structure is used.

- **:make-array** — If the structure being defined is implemented as an array, you can use this option to control those aspects of the array not otherwise constrained by **defstruct**. For example, you might want to control the area in which the array is allocated. Also, if you are creating a structure of type **:array-leader**, you almost certainly want to specify the dimensions of the array to be created, and you may want to specify the type of the array. The **:make-array** option can only be used if the **:type** option was used to specify an Explorer extension type.

  The argument to the **:make-array** option should be a list of alternating keyword symbols for the **make-array** function (see Section 7, Arrays) and forms whose values are the arguments to those keywords. For example, the following form requests that the array be allocated in a particular area:

```
(defstruct (some-structure (:type :array-leader)
                           (:callable-constructors nil)
                           (:make-array (:area 'permanent-storage-area
                                         :dimensions '(4 4))))
          slot-name1 slot-name2)
```

  The **defstruct** macro overrides any of the **:make-array** options that it needs to. For example, if your structure is of type **:array**, then **defstruct** supplies the size of the array regardless of what you specify for the **:make-array** option. If you use the **:initial-element** option to **make-array**, all the slots are initialized, but **defstruct**'s own initializations are performed afterward. If a subtype has been specified for or deduced by **defstruct**, this subtype overrides any **:type** keyword in the **:make-array** argument.

  Noncallable constructors (macros) for structures implemented as arrays recognize the keyword argument **:make-array**. Attributes supplied therein override any **:make-array** option attributes supplied in the original **defstruct** form. If an attribute appears in neither the invocation of the constructor nor in the **:make-array** option to **defstruct**, then the constructor chooses appropriate defaults. The **:make-array** option can only be used with the default style of constructor that takes keyword arguments.

The following example uses the preceding `defstruct` definition; particularly note that the `:callable-constructors` option must be **nil**:

```
(make-some-structure :slot-name1 some-value
                     :slot-name2 another-value
                     :make-array '(:dimensions '(8 8)
                                   :element-type 'fixnum
                                   :initial-element (some-computed-number)))
```

If a structure is of type **:array-leader**, you should specify the dimensions of the array. The dimensions of an array are given to **make-array** as a position argument rather than as a keyword argument, so there is no way to specify them in the previously mentioned syntax. To solve this problem, you can use the keyword **:dimensions** or the keyword **:length** (which both mean the same thing) with a value that is acceptable as **make-array**'s first argument.

- **:size-macro** — This option defines a macro whose expansion is an integer equal to the size of this structure. The exact meaning of the size varies, but generally you need to know this number when you are going to allocate one of these structures yourself (for example, the length of the array or list). The argument of the **:size-macro** option is the name to be used for the macro. If this option is present without an argument, then the macro name is produced by concatenating the name of the structure with the suffix **-size**. For example:

```
(defstruct (doodle :conc-name :size-macro)
   c b)
(macroexpand '(doodle-size)) => 2
```

- **:size-symbol** — This option is like **:size-macro** but defines a global variable rather than a macro. The size of the structure is the variable's value. Using **:size-macro** is considered clearer than using **:size-symbol**.

- **:but-first** — The argument to this option is an access function from another structure, and this structure is not expected to be found outside of the resulting slot from the access function. Actually, you can use any one-argument function or a macro that acts like a one-argument function. Using the **:but-first** option without an argument produces an error. The following example shows how to use this option correctly:

```
(defstruct (head (:type :list)
                 (:default-pointer person)
                 (:conc-name nil)
                 (:but-first person-head))
   nose
   mouth
   eyes)
```

The accessors expand as follows:

```
(nose x) ==> (car (person-head x))
(nose) ==> (car (person-head person))
```

- **:callable-accessors** — This option controls whether accessors are actually functions and therefore *callable* or whether they are actually macros. If this option is given an argument of true, is given no argument, or is unspecified, then the accessors are actually functions. Specifically, they are
  substs, so they have all the efficiency of macros in compiled programs while still being function objects that can be manipulated (passed to **mapcar** and so forth). If the argument is **nil**, then the accessors are actually macros.

- **:callable-constructors** — This option controls whether constructors are actually functions and therefore *callable* or if they are macros. An argument of true makes them functions; an argument of **nil** makes them macros. The default is **t**.

- **:property** — For each structure defined by **defstruct**, a property list is maintained for the recording of arbitrary properties about that structure. (That is, there is one property list per structure definition, not one for each instance of the structure.)

  The **:property** option can be used to give a **defstruct** an arbitrary property. The form (**:property** *property-name value*) gives the **defstruct** a *property-name* of *value*. Neither argument is evaluated. To access the property list, you must look inside the **sys:defstruct-description** structure itself.

- **:print** — This option controls the printed representation of a structure in a way independent of the Lisp dialect in use. For example:

  ```
  (defstruct (doodle :named
                     (:print "#<Doodle -S -S>"
                            (doodle-dee doodle) (doodle-dum doodle)))
     dee
     dum)
  ```

  Of course, this form works only if you use a named type so that the system can recognize examples of this structure automatically.

  The arguments to the **:print** option are used as arguments to the **format** function (except for the stream, of course). They are evaluated in an environment in which the name symbol of the structure (`doodle` in this case) is bound to the instance of the structure to be printed.

  This option works by generating a **defselect** that creates a named structure handler. Do not use the **:print** option if you define a named structure handler yourself because the two named structure handlers will conflict.

- *type* — In addition to the options previously discussed, you can also use any currently defined type (any legal argument to the **:type** option) as an option. This feature is provided mostly for compatibility with the old version of **defstruct**. This option allows you to simple specify *type* instead of (**:type** *type*). This option takes no arguments.

- *other* — You can also specify any valid **defstruct** keyword for the type of structure being defined, provided that this option is specified in the form (*option-name value*). This option is treated exactly like (**:property** *option-name value*). That is, the **defstruct** is given an *option-name* property of *value*.

  This option provides a primitive way for you to define your own options to **defstruct**, particularly in connection with user-defined types. Several of the options previously discussed are actually implemented using this mechanism, including **:times**, **:subtype**, and **:make-array**.

  The valid **defstruct** keywords for a particular type are in a list in the **defstruct-keywords** slot of the **defstruct-type-description** structure for *type*.

Byte Fields

10.5 The byte-field feature of **defstruct** allows you to specify that several slots of your structure are to be bytes that should be packed together in a single integer, 25 bit maximum. Obviously, one advantage of this feature is a more compact structure object, but under certain circumstances, you can manipulate these structures faster with an alterant macro.

The form of a packed slot descriptor is a list in which each element is a slot descriptor that contains a byte specifier (or byte-spec). The syntax for such a slot descriptor is as follows:

({(*slot-name byte-spec* [*init-form* {*slot-option*}*])}+)

The *slot-name*, *init-form*, and *slot-option* are the same as those explained previously. The *byte-spec* defines a field within an integer. The **byte** function is the simplest way to define a slot field. However, the form that you supply for *byte-spec* is evaluated each time the associated *slot-name* is accessed. Thus, you can supply a function for a *byte-spec* that returns different values so that the field moves around within the integer. If *byte-spec* is the symbol **nil**, then the corresponding *slot-name* is defined to refer to the entire integer being allocated for this packed slot descriptor.

Constructors (both functions and macros) initialize words divided into byte fields as if they were deposited in the following order:

1. Initializations for the entire word given in the **defstruct** form
2. Initializations for the byte fields given in the **defstruct** form
3. Initializations for the entire word given in the constructor invocation
4. Initializations for the byte fields given in the constructor invocation

Alterant macros work similarly: the modification for the entire Lisp object is performed first, followed by modifications to specific byte fields. If any byte fields being initialized or altered overlap each other, the action of the constructor and alterant is unpredictable. Consider the following example:

```
(defstruct (phone-book-entry (:type list)
                             (:conc-name nil))
       name
       location
       ((area-code (byte 10. 10.) 512)
        (exchange (byte 10. 0)))
       line-number)

(setf pbe (make-phone-book-entry
           :name "TI Customer Support"
           :location "Austin, Texas"
           :area-code 512.
           :exchange 250.
           :line-number 6179.))
=> ("TI Customer Support" "Austin, Texas" 524538 6179)

(area-code pbe) => 512

(macroexpand '(area-code pbe))
=> (LDB (BYTE 10. 10.) (NTH 2 PBE))
```

Note in the expansion that the accessor function evaluates the byte spec. The compiler optimizes out this evaluation if the byte spec resolves to a constant.

Named Structure Handlers

10.6 Because structures that define new data types are recognizable, they can define generic operations and specify how to handle them. A few such operations are defined by the system and are invoked automatically from well-defined places. For example, **print** automatically invokes the **:print-self** operation if you give it a named structure. Thus, each structure type can define how it should print. The standard defined structure operations are listed in this paragraph. You can also define new structure operations and invoke them by calling the structure as a function just as you would invoke a flavor instance.

Operations on a named structure are all handled by a single function, which is found as the **named-structure-invoke** property of the structure type symbol. It is permissible for a named structure type to have no handler function. In such a case, invocation of any operation on the named structure returns **nil**, and system routines such as **print** take default actions.

If a handler function exists, it is given the following arguments:

- *operation* — The name of the operation being invoked, usually a keyword.
- *structure* — The structure being operated on.
- *additional arguments* — Any other arguments that are passed when the operation is invoked. The handler function should have a rest parameter so that it can accept any number of arguments.

The handler function should return **nil** if it does not recognize the *operation*. The following are the structure operations currently in use:

- **:which-operations** — This operation should return a list of the names of the operations handled by the function. Every handler function must handle this operation, and every operation that the function handles should be in this list.

- **:print-self** — This operation should output the printed representation of the named structure to a stream. The additional arguments are the stream to which output is to be sent, the current depth in list structure, and the current value of ***print-escape***. If **:print-self** is not in the value returned by **:which-operations** or if there is no handler function, **print** uses the #S syntax.

- **:describe** — This operation is invoked by **describe** and should print a description of the structure to ***standard-output***. If there is no handler function or if **:describe** is not in the structure's **:which-operations** list, **describe** prints the names and values of the structure's fields as defined in the **defstruct**.

- **:sxhash** — This operation is invoked by the **sxhash** function and should return a hash code to use as the value of **sxhash** for this structure. It is often useful to call **sxhash** on some (perhaps all) of the slots of the structure and combines the results in some way.

  This operation takes one additional argument: a flag indicating whether it is permissible to use the structure's address in forming the hash code. For some kinds of structure, there may not be a way to generate a good hash code except by using the address. If the flag is **nil**, the system must do the best it can, even if that means always returning zero.

  It is permissible to return **nil** for **:sxhash**. In this case, **sxhash** produces a hash code in its default fashion.

- **:fasd-fixup** — This operation is invoked by **fasload** on a named structure that has been created from data in an object file. The purpose of the operation is to give the structure a chance to clean itself up if, in order to be valid, it needs to have contents that are not exactly identical to those that were dumped. For example, readtables push themselves onto the list **sys:*all-readtables*** so that they can be found by name.

  For most kinds of structures, it is acceptable not to define this operation at all (so that it returns **nil**).

Consider the following example:

```
(defun (:property boat sys:named-structure-invoke)
        (operation struct &rest args)
   (case operation
        (:which-operations '(:describe))
        (:describe
           (format (car args)
                "This sailboat is ~d feet long and ~d feet wide."
                 (boat-length-over-all struct)
                 (boat-beam struct)))
   ))
```

Note that the handler function of a structure type is *not* inherited by other named structure types that include it. For example, the previous definition of a handler for `boat` has no effect at all on the `sailboat` structure. If you need such inheritance, you must use flavors rather than typed structures (see Section 19, Flavors).

Structure Functions

10.7 The following functions operate on structures.

describe-defstruct *instance* &optional *name* Function

This function takes an *instance* of a structure named by *name* and displays a description of the instance, including the values of each of its slots. The *name* argument is optional only if *instance* is an instance of a named structure; if it is unnamed, the function returns an error message if *name* is not provided. The reason for this error message is that **describe-defstruct** must have some way to know which structure *instance* is an instance of. If *instance* is a named structure, then the structure name is already embedded within *instance*, and **describe-defstruct** knows where to find it; therefore, *name* is optional only in this case.

This function prints out the information of a structure in the particular manner shown in the following example; however, you can define your own function to print the information of a named structure in a format of your own choosing.

Suppose an instance of a structure called `player` had been constructed as follows:

```
(setf sir-slam (make-player :name "Darryl Dawkins"
                            :team "New Jersey Nets"
                            :position "C"
                            :games 1
                            :points 12
                            :rebounds 13
                            :assists 2
                            :pts-per-game 12.0))
```

Then, a call to **describe-defstruct** would produce the following:

```
(describe-defstruct sir-slam)
;;; The following information is printed to *standard output*.
 #S<PLAYER 12345678> is a structure of type PLAYER

 name:            "Darryl Dawkins"
 team:            "New Jersey Nets"
 position:        "C"
 games:           1
 points:          12
 rebounds:        13
 assists          2
 pts-per-game:    12.0

#S(player :name "Darryl Dawkins" :team "New Jersey Nets" :position
     "C" :games 1 :points 12 :rebounds 13 :assists 2 :pts-per-game
     12.0)
```

named-structure-p *x* Function

This semipredicate returns **nil** if *x* is not a data type specified by **defstruct**; otherwise, it returns the named structure symbol of *x*.

make-array-into-named-structure *array* Function

With this function, the *array* argument is marked as a named structure and is returned as a named structure. This function is used by **make-array** when creating named structures. You should not normally call it explicitly.

sys:named-structure-invoke *operation instance* &rest *args* Function

This function invokes a named structure operation on *instance*. The *operation* argument should be a keyword symbol, and *instance* should be a named structure. The handler function of the named structure symbol, found as the value of the **named-structure-invoke** property of the symbol, is called with appropriate arguments.

If the structure type has no **named-structure-invoke** property, **nil** is returned.

The form (send *instance operation args* ...), where *instance* is a named structure, has the same effect by calling **named-structure-invoke**.

See also the **:named-structure-symbol** keyword to **make-array** in Section 7, Arrays.

The sys:defstruct-description Structure

10.8 This paragraph discusses the internal structures used by **defstruct** that might be useful to programs that want to interface to **defstruct** nicely. For example, if you want to write a program that examines structures and displays them the way **describe** and the Inspector do, your program should work by examining these structures. The information in this paragraph is also necessary for programmers who are thinking of defining their own structure types.

Whenever the user defines a new structure using **defstruct, defstruct** creates an instance of the **sys:defstruct-description** structure. This structure can be found as the **sys:defstruct-description** property of the name of the structure; it contains such useful information as the number of slots in the structure, the **defstruct** options, and so on.

The following is a simplified version of the way the **sys:defstruct-description** structure is defined. It omits some slots whose meanings are not worth documenting here. (The actual definition is in the SYSTEM package.)

```
(defstruct (defstruct-definition
               (:type list)
               (:default-pointer description))
           name
           size
           property-alist
           slot-alist)
```

The `name` slot contains the symbol supplied by the user to be the name of the structure, such as `yacht` or `phone-book-entry`.

The `size` slot contains the total number of slots in an instance of this kind of structure. This is *not* the same number as that obtained from the **:size-macro** option to **defstruct**. A named structure, for example, usually uses up an extra location to store the name of the structure, so the **:size-macro** option produces a number one larger than that stored in the **defstruct** description.

The `property-alist` slot contains an association list with pairs of the form (*property-name* . *property*) containing properties placed there by the **:property** option to **defstruct** or by property names used as options to **defstruct**.

The `slot-alist` slot contains an association list of pairs of the form (*slot-name . slot-description*). A *slot-description* is an instance of the **defstruct-slot-description** structure. The **defstruct-slot-description** structure is defined something like the following (but also contains slots omitted here) and is also in the SYSTEM package:

```
(defstruct (defstruct-slot-description
               (:type list)
               (default-pointer description))
           number
           PPSS
           init-code
           type
           property-alist
           ref-macro-name
           documentation)
```

The `number` slot contains the number of the location of this slot in an instance of the structure. Locations are numbered starting with 0 and continuing up to a number one less than the size of the structure. The actual location of the slot is determined by the reference-consing function associated with the type of structure.

The `PPSS` slot contains the byte specifier code for this slot if this slot is a byte field of its location. If this slot is the entire location, then the `ppss` slot contains **nil**.

The `init-code` slot contains the initialization code supplied for this slot by the user in the **defstruct** form. If there is no initialization code for this slot, then the `init-code` slot contains canonical objects that can be tested using `(sys:emptyp slot-value)`.

The `ref-macro-name` slot contains the symbol that is defined as a macro or a subst that expands into a reference to this slot (that is, the name of the accessor function).

11

HASH TABLES

Hash Table Definitions

11.1 The data type **hash-table** defines a Lisp object which facilitates accessing data based on an associated key. Like a property list or an association list, a *hash table* associates *keys* with *values*. However, hash tables are much faster for large collections of data because they do not use searching operations to find the value associated with a particular key.

The process of *hashing* computes a *hash code*, a nonnegative integer for each key indicating the location of its associated value. Thus, when a particular key is specified as an argument to a hash table function, the function uses the hash code to index the appropriate mapped location. A check is made (using **eq**, **eql**, or **equal**) to verify that the key in the hash table agrees with the key specified as the argument to the function. If these two keys are the same according to the predicate being used, the hash table function performs its operation. If they are not the same, a *collision* occurs, which means that two or more keys map to the same location.

The size of a hash table indicates how many entries it can hold. When the table's *threshold* is exceeded, the table's size is increased and the entries are *rehashed* automatically. That is, new hash codes are computed and the entries are rearranged according to these new codes. This rehashing process is performed transparently to the caller.

Hash table keys need not be symbols: they can be any kind of Lisp object. Similarly, hash table values can be any kind of Lisp object. Because **eq** does not work reliably on numbers, they should not be used as keys in an **eq** hash table. Use an **eql** hash table if you want to hash on numeric values.

The functionality provided by hash tables can be included in a flavor definition. See paragraph 19.11, Hash Table Operations, for details.

Hash Table Functions

11.2 The following functions perform the basic hash table operations.

make-hash-table &key :test :size :rehash-size :rehash-threshold *[c]* Function
make-hash-table &key :test :size :rehash-size :rehash-threshold :number-of-values Function

This function creates and returns a new hash table. Equality tests other than eql can be used through the keyword :test.

:test — This argument specifies a function that identifies the kind of hashing to be performed. This argument must be a symbol or the function for **eq**, **eql**, or **equal**. The default for this argument is **eql**.

:size — This argument sets the initial size of the hash table, in entries. The actual size is rounded up from the size you specify to the next size that is appropriate for the hashing algorithm. The number of entries you can actually store in the hash table before it is rehashed is at least the actual size times the rehash threshold. On the Explorer system, the default for **:size** is 64.

:rehash-size — This argument specifies the amount to increase the size of the hash table when it becomes full. This can be an integer indicating the number of entries to add, or it can be a floating-point number indicating the ratio of the new size to the old size. On the Explorer, the default is 1.3, which causes the table to be made 30 percent bigger each time it has to grow.

:rehash-threshold — This argument sets a maximum fraction of the entries that can be in use before the hash table is made larger and rehashed. The default is 0.750. Alternately, you can specify an integer, which is the exact number of filled entries at which a rehash should be done. If, when the rehash happens, the **:rehash-threshold** argument is set to an integer, it is increased in the same proportion as the table has grown.

:number-of-values — This argument specifies how many values to associate with each key. The default for this argument is 1. Note that the third value returned by **gethash** is the complete list of keys and values. This keyword is an Explorer extension.

For example:

```
(make-hash-table :rehash-size 15 :test #'eq
                 :size (* number-of-batting-stats 25))
```

hash-table-p *object* *[c]* Function

This predicate evaluates to true if *object* is a hash table; otherwise, it returns **nil**. Note the following equivalence:

```
(hash-table-p object) <=> (typep object 'hash-table)
```

hash-table-rehash-size *hash-table* *[c]* Function
hash-table-rehash-threshold *hash-table* *[c]* Function
hash-table-size *hash-table* *[c]* Function
hash-table-test *hash-table* *[c]* Function

These functions return the appropriate information for the specified *hash-table*.

gethash *key hash-table* &optional *default* *[c]* Function

This function finds the entry in *hash-table* for *key* and returns three values. The first two values are defined by the Common Lisp standard, while the third value is an Explorer extension. The first returned value is the (first) associated entry for *key*, or **nil** if there is no entry. The second value is true if there is an entry for *key* or **nil** if there is not.

On the Explorer system, the third value returned is a list whose car is *key* and whose cdr is all the values associated with *key*. This result allows you to retrieve values other than the first if the hash table has more than one value per entry.

While a **gethash** operation is processing, other processes are *locked out* from access to the hash array to prevent conflicting operations; that is, two processes can safely share a hash table.

You can also use the **setf** macro with **gethash** to add new entries to a hash table. When entries are replaced this way, the original value is removed from the hash table before the new value is added, as you would expect. Also, when **setf** is used with **gethash**, the *default* argument is ignored; however, this argument can be helpful when **gethash** is used in conjunction with macros related to **setf**, such as **incf**.

puthash *key value hash-table* &rest *extra-values* Function

This function is used to add an entry to *hash-table* by associating *key* to *value*. If an entry already exists for *key*, then this function replaces the current value of this key with *value* and returns *value*. The hash table automatically grows if necessary.

If the hash table associates more than one value with each key, the remaining values in the entry are taken from *extra-values*. While a **puthash** operation is processing, other processes are *locked out* from access to the hash array to prevent conflicting operations; that is, two processes can safely share a hash table. Note the following equivalence:

```
(puthash key value hash-table)
<=> (setf (gethash key hash-table) value)
```

remhash *key hash-table* *[c]* Function

This function removes any entry for *key* in *hash-table*. This function returns a true value if there was an entry or **nil** if there was not. Because of these returned values, the function can also be used as a predicate.

maphash *function hash-table* *[c]* Function
maphash *function hash-table* &rest *extra-args* Function

This function applies *function* to each occupied entry in *hash-table*. The arguments passed to *function* include the key of the entry, all the values of the entry, and (as an Explorer extension) any *extra-args*. The **maphash** function always returns **nil**. If the hash table has more than one value per key, all the values, in order, are supplied as successive arguments. Consider the following example:

```
;; Remove all players with batting averages less than .200 from
;; player-hash-table and place them in the list minor-leaguers.
 (setf minor-leaguers ())
 (maphash #'(lambda (player-name batting-average)
               (when (< batting-average 0.200)
                  (push player-name minor-leaguers)
                  (remhash player-name player-hash-table)))
            player-hash-table)
```

Note that if *function* modifies the hash table while the **maphash** is in progress, the results are unpredictable because this could cause the hash table to grow and rehash. The only exception is that *function* can call **remhash** (or **setf** of **gethash**) on the key entry currently being operated on.

maphash-return *function hash-table* Function

This function is similar to **maphash** but accumulates and returns a list of all the values returned by *function* when it is applied to entries in the hash table.

clrhash *hash-table* *[c]* Function

This function removes all the entries from *hash-table*. The value returned is the hash table itself.

swaphash *key value hash-table* &rest *extra-values* Function

This function specifies new value(s) for *key* as does **puthash** but returns values describing the previous state of the entry, exactly like **gethash**. In particular, **swaphash** returns the previous (replaced) associated value as the first value (**nil** if there was none), a true value as the second value if the entry existed previously, and, as the third value, a list whose car is *key* and whose cdr is the list of previous values.

modify-hash *key hash-table-function* &rest *additional-args* Function

This function passes the value associated with *key* in the table to *hash-table-function*; whatever *hash-table-function* returns is stored in the table as the new value for *key*. Thus, the hash association for *key* is both examined and updated according to *hash-table-function*.

The arguments passed to *hash-table-function* are *key*, the value associated with *key*, a flag (which is true if *key* is actually found in the hash table), and the *additional-args* that you specify.

If the hash table stores more than one value per key, only the first value is examined and updated.

hash-table-count *hash-table* *[c]* Function

This function returns the number of filled entries in *hash-table*. If the hash table has just been made or cleared, this function returns 0.

sxhash *object* *[c]* Function
sxhash *object* &optional *ok-to-use-address-p* Function

This function is a primitive that performs the hashing process. The **sxhash** function computes the hash code of *object*, and this hash code is returned as a positive fixnum. With **sxhash**, `(equal x y)` always implies `(= (sxhash x) (sxhash y))`.

This function computes the hash code in such a way that common permutations of an argument, such as interchanging two elements of a list or changing one character of a string, always change the hash code.

TYPE SPECIFIERS

12

Type Specifier Definitions

12.1 *Type specifiers* are Lisp objects that identify data types. These specifiers come in two different forms: a symbol or a type-specifier list.

In practice, type specifiers have two fairly distinct uses: declaration and discrimination. *Declaration* allows the programmer to specify to the system the intended use of a particular variable or array. The purpose of this information is to allow the system to optimize generated code, although in most cases Common Lisp does not require that the optimization be performed. For example, informing **make-array** that an array's element type should be **integer** does not mean that the array being created can only contain integers. It merely tells the system to use the most efficient implementation scheme to hold objects of type **integer**.

On the other hand, *discrimination* means that a program wants to determine if a particular data object is or is not of a specific data type. The simplest example of discrimination is the use of the **typep** function.

Using Type Specifier Lists

12.2 The type specification for a vector is a list of the form (**vector** *type length*). Thus, the following example defines a data type of vectors with 150 elements of type **short-float:**

```
(vector short-float 150)
```

You can leave one of these restrictive elements unspecified by using an asterisk:

```
(vector short-float *)
```

In this case, the vector can be of any length. Furthermore, the type specifier `(vector short-float 150)` is a subtype of `(vector short-float *)` because the latter type specifier includes all vectors whose elements are of type **short-float,** both those with exactly 150 elements and those with more or fewer elements.

For convenience, when a type specifier ends with one or more asterisks, you can omit the trailing asterisks. For instance, the type specifier `(vector short-float *)` can be abbreviated as `(vector short-float)`. If, as a result of omitting the asterisks, the type specifier list is reduced to a type name, then that type can be represented by that type name. For instance, `(vector * *)` is equivalent to `vector`. Table 12-1 (in paragraph 12.5, Type Specifier Symbols) includes all the standard Common Lisp type specifiers that have a single symbol name as an abbreviation.

Basic Type Specifiers

12.3 Most data types are defined in the section of this manual which describes that type of object. For example, number types are described in Section 3, Numbers. The following type specifiers are of a more general nature (they did not fit in any particular section), or are of a more complex nature such that the specification may have restrictive arguments. When any of the arguments are explicitly supplied, then the type specifier should be represented as a list.

atom Type Specifier

This type specifier represents all objects that are not conses.

common Type Specifier

This type specifier represents all objects whose types are specified by Common Lisp. For example, objects whose types are defined in Table 12-2 are not Common Lisp objects. (Table 12-2 appears in paragraph 12.5, Type Specifier Symbols.)

keyword Type Specifier

This type specifier represents all objects that are symbols in the KEYWORD package.

nil Type Specifier

This type specifier is defined to represent no Lisp object. No objects of this type exist.

t Type Specifier

This type specifier is defined to represent all Lisp objects.

array *element-type dimensions* *[c]* Type Specifier

This type specifier represents all objects that are arrays whose rank and dimensions fit the restrictions described by *dimensions* and whose type restricts possible elements to match *element-type*.

The array elements specification has nothing to do with the actual values of the elements. Rather, it is a question of whether the array's own type permits exactly such elements as would match *element-type*. If anything can be stored in the array that does not match *element-type*, then the array is not of this type. If anything that matches *element-type* cannot be stored in the array, then the array is not of this type.

If *element-type* is **t**, the type to which all objects belong, then the array must be one in which any object can be stored: **art-q** or **art-q-list**.

If *element-type* is * (meaning *no restriction*), any type of array is then allowed, whether it restricts its elements or not.

The *dimensions* argument can be *, an integer, or a list. If it is *, the rank and dimensions are not restricted. If *dimensions* is an integer, it specifies the rank of the array. In any case, any array of that rank matches, and the *dimensions* are not restricted.

If *dimensions* is a list, its length specifies the rank, and each element of *dimensions* restricts one dimension. If the element is an integer, that dimension's length must equal that integer. If the element is *, that dimension's length is not restricted.

For example, the following form is a type specifier for four-dimensional arrays containing rational numbers:

```
(array rational 4)
```

The following is a type specifier for a 2-by-20 array containing any kind of objects:

```
(array * (2 20))
```

The following is a type specifier for a two-dimensional array with seven columns and any number of rows; the array elements are integers:

```
(array integer (* 7))
```

simple-array *element-type dimensions* *[c]* Type Specifier

This type specifier is equivalent to **array** except that the array is also a simple array. (See Section 7, Arrays.)

vector *element-type size* *[c]* Type Specifier

This type specifier represents all objects that are vectors of *element-type* and of *size*. The *element-type* argument operates as described in the **array** type specifier mentioned previously. The *size* argument must be an integer or *; if it is an integer, the array's total length, not counting the fill pointer, must equal *size*.

For example, the following form specifies a vector of 12 characters:

```
(vector character 12)
```

The following specifies a vector of any length that can hold any kind of objects:

```
(vector t *)
```

Note that the two previous examples describe mutually exclusive subsets. That is, although `(vector t *)` can hold characters, it is not specifically made to optimal storage of characters, as is `(vector character 12)`. In other words, `(vector character 12)` is not a subtype of `(vector t *)`.

bit-vector *size* *[c]* Type Specifier
simple-vector *size* *[c]* Type Specifier
simple-bit-vector *size* *[c]* Type Specifier
string *size* *[c]* Type Specifier
simple-string *size* *[c]* Type Specifier

These type specifiers require the vector to match type **bit-vector**, **simple-vector**, and so on. The *size* argument works as in vector.

complex *type-spec* *[c]* Type Specifier

This type specifier represents all complex numbers whose components match *type-spec*. Thus, `(complex rational)` represents the type of complex numbers with rational components.

function (*parameter-type-spec*) *return-value-type* *[c]* Type Specifier

This type specifier identifies a function that accepts arguments in accordance with *parameter-type-spec* and whose returned value is of type *return-value-type*. If the function type specified returns multiple values, you can specify them by using the **values** type specifier.

Note that this type specifier can only be used in a **declare** or **proclaim** form and is not suitable for discrimination purposes, such as an argument to **typep**.

A *parameter-type-spec* is a mapping between the data type of a particular argument and the respective parameter within a function's lambda list. Generally speaking, a *parameter-type-spec* looks like a lambda list, and the implied mapping is fairly intuitive. The main difference between a *parameter-type-spec* and a lambda list is that in a lambda list you specify a parameter name, whereas in the *parameter-type-spec* you provide a type specification for the corresponding positional argument. In a *parameter-type-spec*, keywords are identified by the list (*keyword-name keyword-type-spec*). You can use the lambda-list keywords &optional, &rest, &key, and &allow-other-keys to facilitate this mapping. For example, the following describes a function that accepts an array as its first argument, an optional second argument of any type, and, if present, numbers for the remaining arguments. The returned value is a single value of any type.

```
(function (array &optional t &rest number) t)
```

The following type specification describes functions with two positional arguments: the first can be any object, and the second must be a sequence. A function of this type can accept any keyword arguments, but if the **:test** keyword is supplied, its value must be a function. The returned value of this function is a single object whose type is not restricted.

```
(function (t sequence &key (:test function) &allow-other-keys) t)
```

When you describe a function type specification that has keywords, all keywords must be accounted for. If you do not explicitly list them by name, then you must use &allow-other-keys to indicate that other keywords are expected.

values {*value-type*}+ *[c]* Type Specifier

This type specifier is used to identify the type and number of returned values in two cases: when it is used in the *return-value-type* argument to the **function** type specifier, and when it is used in the special form **the**. Note that this type specifier has the same name as the function that produces multiple values, but naturally their purpose is different.

The *value-type* arguments are a sequence of type specifiers, each of which corresponds to the data type of the multiple values being returned. For instance, the function **floor** returns two values, both of which are integers. Thus, you could write the following code:

```
(the (values integer integer)              ; The type specification
   (floor some-dividend some-divisor))     ; The form that produces
                                           ; multiple values
```

The advantage in using this kind of form is that the compiler can now attempt certain optimizations based on this extra information.

The lambda-list keywords &optional, &rest, and &key have a special meaning in the context of the **values** type specifier. They can be used in combination just as they are in a **defun** lambda list, but to keep the explanations simple, the use of each one is described separately here.

The use of the &optional keyword is fairly straightforward. Its syntax is as follows:

```
(values &optional data-type)
```

This specifier means that a function can return no more than one value. A practical use of this feature might be the following:

```
(the (values integer &optional integer)    ; The type specification
   (or some-integer-or-nil                 ; The case of one returned value
      (floor some-dividend some-divisor)))
                                           ; The case of two returned values
```

The use of the &key keyword is fairly obscure. The syntax is as follows:

```
(values &key (a-keyword-name keyword-value-type))
```

This specifier means that the function could return two values: the first should be `:a-keyword-name`, and the second should be a data value whose type is `keyword-value-type`. In practice, this kind of specifier is used to declare that a function is to return multiple values that will subsequently be used as a keyword-and-values pair in a **multiple-value-call** form. For example:

```
(multiple-value-call 'make-string
                      size-of-string
                      (the (values &key (initial-element character))
                           (determine-initial-element)))
```

The function `determine-initial-element` can return no values so that the `initial-element` argument to `make-string` defaults. Alternatively, `determine-initial-element` could return two values, the first of which should be `:initial-element` and the second of which should be an object of type `character`.

The use of the &rest keyword is also obscure. Its syntax is as follows:

```
(values &rest data-type)
```

This specifier means that all returned values that are covered by the &rest argument must be of type `data-type`. Note that if you use &rest and &key together, then the `data-type` should allow for the keyword symbol and value that the &key specification allows.

satisfies *type-predicate* *[c]* Type Specifier

This type specifier specifies a type according to all values for which the functional argument *type-predicate* returns true. The argument *type-predicate*, which is passed only one argument, is the specifying predicate. (This argument can only be a function name; lambda expressions are not allowed because of scoping problems.) For example:

```
(setq x #C(4.03s1 5.31s-2))
(typep x '(satisfies numberp)) => true
```

In this example, `x` is set to a complex number and then used as an argument to the `typep` function along with the predicate-specified type specifier. In this case, `numberp` is applied to `x`, and because `x` is a number, `true` is returned. The following example shows how a new data type can be defined using a predicate-specified type specifier:

```
(deftype even-count ( )
  '(and (satisfies integerp) (satisfies plusp) (satisfies evenp)))
```

This example defines a new data type called `even-count`, which is the set of all objects that are positive even integers. (The **deftype** macro is described in paragraph 12.6, Defining New Type Specifiers.)

CAUTION: The predicate used in a predicate-defined type specifier should not cause side effects when invoked because it is not easy to predict exactly when this predicate will be called.

integer *low high* *[c]* Type Specifier

This type specifier represents all integers between *low* and *high*. When *low* is simply integer *n*, *n* is an inclusive lower limit. When *low* is an integer contained in a list (*n*), *n* is an exclusive lower limit. When *low* is *, there is no lower limit.

The *high* argument has the same possibilities. If *high* is omitted, it defaults to *. If both *low* and *high* are omitted, then this form is equivalent to just `integer`. Consider the following examples:

```
(integer 0)     ; Specifies a nonnegative integer.
(integer -4 3)  ; Specifies an integer between -4 and 3, inclusive.
```

mod *high* *[c]* Type Specifier

This type specifier represents all nonnegative integers less than *high*. The *high* argument should be an integer. The forms `(mod)`, `(mod *)`, and `mod` are allowed but are equivalent to `(integer 0)`.

signed-byte *size* *[c]* Type Specifier

This type specifier represents integers that fit into a byte of *size* bits, where one bit is the sign bit. The type specifier `(signed-byte 4)` is equivalent to `(integer -8 7)`. Also, `(signed-byte *)` and `signed-byte` are equivalent to `integer`.

unsigned-byte *size* — *[c]* Type Specifier

This type specifier represents nonnegative integers that fit into a byte of *size* bits, with no sign bit. The type specifier `(unsigned-byte 3)` is equivalent to `(integer 0 7)`. Also, `(unsigned-byte *)` and `unsigned-byte` are equivalent to `(integer 0)`.

rational *low high* — *[c]* Type Specifier
float *low high* — *[c]* Type Specifier
short-float *low high* — *[c]* Type Specifier
single-float *low high* — *[c]* Type Specifier
double-float *low high* — *[c]* Type Specifier
long-float *low high* — *[c]* Type Specifier

These type specifiers indicate the restrictive bounds *low* and *high* for the types **rational**, **float**, **short-float**, and so on. The bounds work on these types the same way they do on **integer**.

Type Specifiers That Combine

12.4 The following type specifiers define a data type consisting of a combination of other data types or objects.

member {*object*}* — *[c]* Type Specifier

This type specifier represents all objects that are **eql** to any one of *object*s. Thus, the following is matched only by `t`, `nil`, or `x`:

```
(member t nil x)
```

not *type-spec* — *[c]* Type Specifier

This type specifier represents all objects that are not of the type specifier *type-spec*.

and {*type-spec*}* — *[c]* Type Specifier

This type specifier represents individually all objects that are of all the type specifiers indicated in *type-spec*s. Thus, the following is the type of odd integers:

```
(and integer (satisfies oddp))
```

Testing is done from left to right, so the `oddp` function is not called unless the object is first determined to be an integer.

or {*type-spec*}* — *[c]* Type Specifier

This type specifier represents individually all objects that are of at least one of the type specifiers indicated in *type-spec*s. Thus, the following includes all numbers and all arrays:

```
(or number array)
```

Type Specifier Symbols

12.5 The Explorer system provides Common Lisp type specifier symbols and Explorer extension type specifier symbols. Note that any data type created by **defstruct, deftype,** or **defflavor** is also available as a legitimate type specifier symbol.

Table 12-1 shows the Common Lisp type specifier symbols available on the Explorer system.

Table 12-1

Common Lisp Symbolic Type Specifiers

array
atom
bignum
bit
bit-vector
character
common
compiled-function
complex
cons
double-float
fixnum
float
function
hash-table
integer
keyword
list
long-float
nil
null
number
package
pathname
random-state
ratio
rational
readtable
sequence
short-float
signed-byte
simple-array
simple-bit-vector
simple-string
simple-vector
single-float
standard-char
stream
string
string-char
symbol
t
unsigned-byte
vector

Table 12-2 shows the type specifier symbols that are Explorer extensions.

Table 12-2

Explorer Extension Symbolic Type Specifiers

closure
instance
locative
microcode-function
real
stack-group
structure

Defining New Type Specifiers

12.6 The deftype macro allows you to define your own type specifiers.

deftype *type-name lambda-list* {*declaration*|*doc-string*}* *body* *[c]* Macro

This macro defines *type-name* as a type specifier by providing code to expand it into another type specifier—a kind of type specifier macro.

When a list starting with *type-name* is encountered as a type specifier, the *lambda-list* is matched against the cdr of the type specifier just as the lambda list of an ordinary macro defined by **defmacro** is matched against the cdr of a macro call form. Then the *body* is executed and should return a new type specifier to be used instead of the original form.

If there are optional arguments in *lambda-list* for which no default value is specified, they receive * as a default value.

If *type-name* by itself is encountered as a type specifier, it is treated as if it were (*type-name*); that is, the *lambda-list* is matched against no arguments, and the *body* is executed. In this case, each argument in the *lambda-list* receives its default value, and there is an error if they are not all optional.

If *doc-string* is supplied, it is associated with *type-name* and can be accessed using the **documentation** function with a *doc-type* of `'type`.

Consider the following **deftype** examples:

```
;; This type definition could have been used to define the type "vector".
(deftype vector (element-type size)
            `(array ,element-type (,size)))

(deftype odd-natural-number-below (n)
            `(and (integer 0 (,n)) (satisfies oddp)))

(typep 5 '(odd-natural-number-below 6)) => true

(typep 7 '(odd-natural-number-below 6)) => nil
```

Type Identification and Execution Control

12.7 The following functions and macros are associated with identifying an object's data type. Some of the forms are also associated with control of execution based on type identification.

type-specifier-p *object* — *[c]* Function

This function returns true if *object* is a valid type specifier; otherwise, it returns **nil**. Note that types defined by **deftype**, **defstruct**, and **defflavor** are accepted as valid type specifiers.

type-of *object* — *[c]* Function

In the Common Lisp definition, the value returned by this function depends on the implementation in effect. On the Explorer system, this function returns a symbol corresponding to the machine data type of *object*, such as **fixnum**, **bignum**, **symbol**, **array**, **cons**, and so on. If the argument is a flavor instance, then the name of the flavor is returned. If the argument is a structure instance of a named structure, then the name of the structure is returned.

This function is intended to be used only for debugging information purposes. To test whether an object is of a certain type, use **typep** or **typecase**.

typecase *key-form* {(*type-spec* {*forms*}*)}* — *[c]* Macro

This macro evaluates *key-form* and then executes one (or none) of the clauses according to the type of the value, which will be called *key-form-value*.

Each clause starts with a *type-spec*, not evaluated, which should be acceptable as the second argument to **typep**. (In fact, the **typecase** macro expands to a call to **typep** with *type-spec* as the second argument.) The rest of the clauses are composed of *forms*. The *type-spec* of each clause is matched sequentially against the type of *key-form-value*. If there is a match, the rest of that clause is executed and the value(s) of the last form is returned from the **typecase** form. If no clause matches, the **typecase** form returns **nil**.

Note that **t**, the type specifier that matches all objects, is useful in the last clause of a **typecase**. The **otherwise** form can be used instead of **t** with the same meaning. For example:

```
(typecase foo
   (symbol (symbol-name foo))
   (string foo)
   (list (apply 'string-append (mapcar 'hack foo)))
   ((integer 0) (hack-positive-integer foo))
   (t (princ-to-string foo)))
```

etypecase *key-form* {(*type-spec* {*forms*}*)}* *[c]* Macro

This macro is like **typecase**, except that an uncorrectable error is signaled if every clause fails. Neither the **t** nor the **otherwise** clause is allowed.

ctypecase *place* {(*type-spec* {*forms*}*)}* *[c]* Macro

This macro is like **etypecase**, except that the error is correctable. The first argument is called *place* because it must be a place form acceptable to **setf**. If the user proceeds from the error, a new value is read and stored into *place*; then the clauses are tested again using the new value. Errors repeat until a value is specified that makes some clause succeed.

Type Predicates

12.8 The following predicates use type specifiers to determine if an argument is of a particular type.

typep *object type-spec* *[c]* Function

This predicate is used to test whether objects are of a specified type. This predicate returns true if the type of *object* matches *type-spec*.

Because some types are subtypes of others, an *object* is not necessarily of one type only. The *type-spec* argument can be any type specifier other than the **function** type specifier or the **values** type specifier.

Calling **typep** with only its *object* argument is an obsolete way of specifying **(type-of** *object*). Consider the following examples:

```
(typep 5 'number) => true
(typep 5 '(integer 0 7)) => true
(typep 5 'bit) => nil
(typep 5 'array) => nil
(typep "foo" 'array) => true
(typep nil 'list) => true
(typep '(a b) 'list) => true
(typep 'lose 'list) => nil
```

If the value of *type-spec* is known at compile time, the compiler optimizes **typep** so that it does not decode the argument at run time.

subtypep *type1 type2* *[c]* Function

This predicate returns two values to indicate whether *type1* is a subtype of *type2*. If the first value is true, then *type1* is definitely a subtype of *type2*. If the first value is **nil**, then *type1 may* not be a subtype of *type2*.

The system cannot always tell whether *type1* is a subtype of *type2*. When **satisfies** type specifiers are in use, this question is mathematically undecidable. Because of this, it has not been considered worthwhile to make the system able to answer obscure subtype questions even when it is theoretically possible. If the answer is not known, **subtypep** returns **nil**.

Therefore, **nil** could mean that *type1* is certainly not a subtype of *type2*, or it could mean that there is no way to tell whether it is a subtype. The **subtypep** function returns a second value to distinguish these two situations: the second value is true if the first value returned by **subtypep** is definite, whereas the second value is **nil** if the system does not know the answer. For example:

```
(subtypep 'cons 'list) => true true
(subtypep 'null 'list) => true true
(subtypep '(satisfies oddp) '(satisfies evenp)) => nil nil
(subtypep 'rational 'number) => true true
(subtypep 'number 'rational) => nil true
(subtypep 'list 'number) => nil true
(subtypep 'symbol 'list) => nil true
```

commonp *object* *[c]* Function

This predicate returns true if *object* is of a type that Common Lisp defines operations on; otherwise, it returns **nil**.

Type Conversion

12.9 The **coerce** function allows you to convert an object to a different data type.

coerce *object type-spec* *[c]* Function

This function converts *object* to an equivalent object that matches *type-spec*. Common Lisp specifies exactly which types can be converted to which other types. In general, anything that would lose information, such as turning a floating-point number into an integer, is not allowed as a coercion. The following is a complete list of the types you can coerce to:

- **complex**, (**complex** *type*) — Real numbers can be coerced to complex numbers. If a rational is coerced to type **complex**, the result equals the rational and is not complex at all. This is because complex numbers with rational components are canonicalized to real numbers if possible. However, if a rational is coerced to (**complex single-float**) then an actual complex number does result. It is permissible, of course, to coerce a complex number to a complex type. The real and imaginary parts are coerced individually to *type-spec* if *type-spec* is specified. For example:

  ```
  (coerce 75 'complex) => 75
  (coerce 7.5s0 'complex) => #C(7.5s0 0.0s0)
  (coerce 7.5s0 '(complex short-float)) => #C(7.5s0 0.0s0)
  ```

- **short-float, single-float** — Rational numbers can be coerced to floating-point numbers, and any kind of floating-point number can be coerced to any other floating-point format. For example:

  ```
  (coerce 78 'short-float) => 78.0s0
  (coerce 77/78 'short-float) => 0.98718s0
  (coerce 78 'single-float) => 78.0
  ```

- **float** — Rational numbers are converted to single-float numbers.

- **character** — Strings of length 1 can be coerced to characters. Symbols whose print names have length 1 can be coerced also. Integers can be coerced to characters. For example:

```
(coerce 78 'character) => #\N
(coerce "78" 'character) => ERROR
(coerce "8" 'character) => #\8
```

- **list** — Any vector can be coerced to type **list**. The resulting list has the same elements as the vector. For example:

```
(coerce #(x y z) 'list) => (x y z)
```

- **vector**, **array** or any restricted array type — Any sequence (a list or a vector) can be coerced to any array sequence or vector type. If you specify a type of array that restricts the type of elements it can hold, you can actually produce an array that can hold other kinds of objects. For example, the Explorer system does not provide anything of type **(array symbol)**, but if you specify this type, you get an array that at least can hold symbols (but can hold other things). If an element of the original sequence does not fit in the new array, an error is signaled.

- **t** — Any object can be coerced to type **t**. Actually, no change occurs to the object because all objects are of type **t**.

If the value of *type-spec* is known at compile time, the compiler optimizes **coerce** so that it does not decode the argument at run time.

DECLARATIONS 13

Declaration Definitions

13.1 Declarations are used to supply extra information to the Lisp environment about your Lisp code. For the most part (except with the **special** declaration), the information you supply does not affect your algorithm. For practical purposes, use declarations to document your algorithm to make it more clear, more precise, or more efficient.

Some declarations advise the compiler that certain assumptions can be made, thus allowing particular kinds of optimizations. Other kinds of declarations describe diagnostic conditions that can be used to supply error checking. The Common Lisp standard states that the use of declarative information is completely optional and implementation dependent. It can also be assumed that any Common Lisp program that runs correctly with declarations will also run correctly without those declarations or, alternatively, on a Common Lisp system that does not support those declarations.

The only exception to this rule is a group of special variable declarations. Because they do make a difference to the algorithm, every Common Lisp implementation must adhere to these declarations.

Nonpervasive declaration specifiers pertain only to the variable bindings that are established at the beginning of the declaring form. If a nested form lexically shadows the original variable binding, the original declaration does not affect the new binding. For example:

```
(defun test (x)
   (declare (type string x))
   (let ((x 1))
     ...))
```

The `let` form inside of the `defun` establishes a new variable binding of `x`, which is not affected by the earlier `type` declaration.

Pervasive declaration specifiers are those that have no effect on variable bindings. Rather, the information that they convey pertains to the entire declaring form, including nested forms. For example:

```
(defun test (x)
   (declare (inline my-function))
   (my-function)
   (let ()
     (my-function)))
```

In this example, the `inline` declaration pertains to both calls to `my-function`.

Some forms that use declarations contain peripheral code that is not part of the form's body, for example, initialization forms of a lambda list and the return forms of iteration constructs like **do**. Nonpervasive declarations do not affect the peripheral pieces of code, whereas pervasive declarations do. For example:

```
(defun foo (x &optional (y num))
    (declare (type float num))
     ...))
```

The reference to `num` in the lambda list of the first line of this example is not affected by the declaration in the second line because the declaration specifier is nonpervasive.

Consider the following example:

```
(defun foo (x &optional (y *spvar*))
    (declare (special *spvar*))
     ...)
```

In this example, the reference to `*spvar*` in the lambda list is affected by the declaration in the second line because the `special` declaration pervasively affects all references to `*spvar*` within `foo`.

Declaration Forms

13.2 Declarations can be either *global* or *local*. To make global declarations that affect the entire Lisp environment, use the **proclaim** function. To make local declarations, insert a **declare** statement into one of the special forms listed in the **declare** description, or use the **locally** macro.

declare {*decl-spec*}* *[c]* Special Form

This special form is used to make local declarations within certain forms. Local declarations must appear in specified locations within the forms that use them (normally they appear immediately before the body of the form). The specified locations are noted in each functional description of the forms that use them.

The forms that are permitted to use declarations are lambda expressions and any of the following:

define-setf-method
defmacro
defmethod
defsetf
deftype
defun
defun-method
do
do*
dolist
dotimes
do-all-symbols
do-external-symbols
do-symbols
flet
labels
let
let*
locally
macrolet
multiple-value-bind
prog
prog*
with-input-from-string
with-open-file
with-open-stream
with-output-to-string

These forms explicitly check for the **declare** form and process this declaration form prior to carrying out their intended purpose. Specifically, the evaluator processes a **declare** statement only at the lexical top level of these forms, and it is an error to evaluate a declaration at any other time.

A macro call is permitted to expand into a declaration, provided that the macro call is located where a declaration is supposed to be located. However, it is not permitted for a macro call to be supplied as a *decl-spec* argument because these arguments are not evaluated.

The values specified for the *decl-spec* argument must each be a declaration specifier in the form of a list. The list's first element (which is a symbol) specifies the type of the declaration to be made by **declare**. These specifiers either affect variable bindings (*nonpervasive declaration specifiers*) or do not (*pervasive declaration specifiers*). However, **special** is actually both pervasive and nonpervasive; it affects how referencing within the declaring form works (it specifies to use the dynamic binding), and it affects variable bindings (it makes dynamic bindings).

locally {*decl-spec*}* {*body-form*}* *[c]* Macro

This macro executes *body-form* within the context of the *decl-spec*. This macro is synonymous with the **progn** special form, except that in Common Lisp, **progn** does not allow declarations at the beginning.

Another difference with **progn** is that at the top level in a file being compiled, **progn** causes each of its elements to be treated as if at the top level, but **locally** does not receive this treatment. The **locally** form is simply evaluated when the compiled code is loaded.

proclaim *decl-spec* *[c]* Function

This function makes the *decl-spec* globally effective. The **proclaim** function is a replacement for the obsolete traditional use of **declare** at the top level. (In Common Lisp, **declare** is used only for local declarations.) The **proclaim** function is different from **declare** in that it is a function, and its arguments are evaluated when it is called. Therefore, the arguments to **proclaim** must be quoted if it is a declaration specifier, and if it is a variable or a call form, the returned value must be a declaration specifier. For example, at the top level you write the following:

```
(proclaim '(special x))
```

Top-level **special** declarations are not the recommended way to make a variable special. Use **defvar, defconstant,** or **defparameter** so that you can give the variable documentation. Proclaiming the variable special should be done only when the variable is used in a file other than the one that defines it. This convention allows the file to be compiled without having to load the defining file first.

special {*variable*}* Special Form

This special form declares each *variable* to be globally special. When you are declaring globally special variables, it is usually better to use **defparameter** or **defvar**. This special form is considered obsolete and is equivalent to the following:

(**proclaim** '(**special** *variables*))

unspecial {*variable*}* Special Form

This special form removes any special declarations of the *variables*. This special form is obsolete and is equivalent to the following:

(**proclaim** '(**unspecial** *variables*))

Declaration Specifiers

13.3 The following are the Common Lisp declaration specifiers.

special *{var}*+ *[c]* Declaration Specifier

For this declaration specifier, the *var* variables are treated as special variables within the scope of the declaration. The **special** declaration specifier is nonpervasive with regard to binding and pervasive with regard to referencing. Thus, if you bind a variable in a form and declare it special, a nested form can create another binding to shadow the first variable such that the new variable is not special (unless explicitly declared to be so in the nested form). For example:

```
(let ((x "special value"))
   (declare (special x))
   (list (symbol-value 'x)
         (let ((x "local value"))
            (symbol-value 'x))))
=> ("special value" "special-value")
```

Recall that the `symbol-value` function returns the current special value. Note that the special binding of `x` was not affected by the local declaration within the `let`.

On the other hand, the pervasive aspect of the **special** declaration specifies that references to a variable appearing in this declaration will access the current dynamic binding (not the current local binding). For example:

```
(setq x "special value")
(let ((x "local value"))
   (list x
          (locally
              (declare (special x))
              x)))
=> ("local value" "special value")
```

Note that if `x` had been proclaimed globally special by a **defvar** or **defparameter**, the `let` would have created a special binding, in which case the returned value would have been `("local value" "local value")`. This difference occurs because **proclaim** has a pervasive effect on binding whereas **declare** does not.

For this reason, it is important to keep track of those symbols that are globally special. It is conventional to begin and end special variable names with asterisks, though no part of the system requires it.

unspecial *{var}*+ Declaration Specifier

The *var* variables are treated as lexical variables within the scope of the declaration, even if they are globally special.

type *type-specifier {var}*+ *[c]* Declaration Specifier

This declaration specifier is a nonpervasive declaration that affects variable bindings only. The variables must take on values of type *type-specifier*.

This specifier can be abbreviated by writing (*type-specifier var1 var2* ...), provided that *type-specifier* is one of the system-defined type specifier symbols listed in paragraph 12.5, Type Specifier Symbols.

It is an error for two function type declarations to refer to the same lexical binding. In practice, this means that it is an error if a variable name appears in more than one type declaration per set of declarations. For example, in a **let** form a variable should have its type declared only once even if one type is a subtype of the other. You can proclaim the type of a global variable as often as you want, in which case the most recent proclamation supersedes all others.

ftype *function-type {function-name}+* *[c]* Declaration Specifier

This declaration pertains only to the bindings of the *function-names* and specifies that the values they take on are only of type *function-type*. The *function-type* argument can be any valid function type specifier. The **ftype** declaration observes lexical scoping rules; thus, for any lexically apparent local definition for *function-name*, the **ftype** declaration pertains to the local definition and not to the global definition. For example:

```
(flet ((first (x) (car x)))
   (declare (ftype (function (cons) t) first))
...)
```

Note that in this example the form `(function (cons) t)` is a type specifier; try not to confuse it with the declaration form of the same name.

It is an error for two function type declarations to refer to the same lexical binding. In practice, this means that it is an error if a variable name appears in more than one function type declaration per set of declarations. You can proclaim the type of a global function as often as you want, in which case the most recent proclamation supersedes all others.

function *name arglist return-value-type* *[c]* Declaration Specifier

This declaration specifier provides the same functionality as **ftype** except that only one function name is allowed in this syntax. As with **ftype**, multiple return values can be expressed using the **values** type specifier. This declaration is sometimes preferred because it is simpler to write and because it resembles the **defun** syntax. For example:

```
(flet ((first (x) (car x)))
   (declare (function first (cons) (values t)))
...)
```

This form is equivalent to the example for **ftype** above.

inline *{function-symbol-spec}+* *[c]* Declaration Specifier
inline *{function-spec}+* Declaration Specifier

With this specifier, the function specs are open-coded or optimized by the compiler within the scope of the declaration, but the compiler can choose to disregard this declaration. This pervasive declaration specifier can be used to increase the execution speed of a function, but the trade-off is that code size usually increases and an open-coded function's ability to be debugged is decreased because the inline function cannot be traced. On the Explorer, inline declarations are implemented in most cases, provided that an interpreted definition of the function is available.

The **inline** declaration observes lexical scoping rules; therefore, if a lexically apparent definition of one of the function specs is defined (via **flet** or **labels**), then the **inline** declaration applies to that local definition and not to the global definition. If a new nested lexical definition for the named function is defined, as with an **flet**, it is not treated as **inline** unless the **flet** declares it to be so.

It is an error for an **inline** and **notinline** declaration to refer to the same function spec within the same set of declarations. You can proclaim a global function **inline** or **notinline** as often as you want; the most recent proclamation supersedes all others.

Note that the only function spec that Common Lisp defines is a symbol name. The use of function specs other than symbols is allowed only as an Explorer extension.

Also note that functions defined by **defsubst** or **defstruct** are expanded inline by default and that an **inline** declaration has no effect on macros or on special forms that the compiler handles specially.

notinline *{function-symbol-spec}+* *[c]* Declaration Specifier
notinline *{function-spec}+* Declaration Specifier

With this declaration, the function specs are not open-coded or optimized by the compiler within the scope of the declaration. The compiler cannot choose to disregard this declaration.

A **notinline** declaration causes calls to the function to be compiled into code that actually calls the function as written, preventing the compiler from doing any of the following:

- Expanding the function inline in response to an outer-level **inline** declaration
- Expanding inline a function defined by **defsubst** or **defstruct**
- Optimizing the call to use a different function or a modified argument list
- Using an equivalent machine instruction instead of a function call

A **notinline** declaration has no effect on macros or on most of the predefined special forms.

Note that the rules of lexical scoping are followed: if one of the functions within this declaration has a local definition (made by such forms as **flet** or **labels**), then the declaration affects the local function definition and not the global function definition.

Also note that the only function spec Common Lisp defines is a symbol name. The use of function specs other than symbols is allowed only as an Explorer extension.

ignore *{var}+* *[c]* Declaration Specifier

The purpose of this declaration is to inform the compiler not to issue a warning message about a variable being unused. This specifier states that the variables, which are bound in the form that uses this declaration, are intentionally not referenced in the body of the form.

optimize {(*feature value*) | *feature*}+ *[c]* Declaration Specifier

This declaration allows you to specify the importance of each *feature*. These *features* are symbols that refer to various aspects of compiler optimization; these are the standard features:

- **speed** — Execution speed of the object code
- **space** — Memory size of the object code
- **safety** — Error checking and ease of debugging
- **compilation-speed** — Speed at which object code is compiled

Each feature is given a corresponding integer value, *value*, indicating the importance of that feature. Each *value* must be between 0 and 3 (inclusive), with 3 being the value of greatest importance. Note that several features can be given the same value. In fact, the default value for all features is 1. To set a feature to its maximum value, you can simply specify *feature* rather than (*feature* 3). This declaration is pervasive. Consider this example:

```
(defun road-runner (a b)
  (declare (optimize (safety 0)))
  (error-check a b)
  (setup a)
  (locally
    ;; This inner loop needs to execute at maximum speed.
   (declare (optimize speed))
    (do
      ...
       )))
```

If **speed** is specified as more important than **space**, then optimizations are enabled that minimize execution time at the expense of increasing the size of the code.

If **compilation-speed** is more important than **speed** or **space**, then some optimizations that slow down compilation are not performed. However, the difference in compiler speed may not be enough to be noticeable.

If **safety** is most important, then some optimizations that make debugging more difficult are prevented. Specifying `(safety 0)` allows some additional optimizations that either complicate debugging (such as tail recursion elimination) or that create new dependencies between modules (such as automatic inline expansion of short functions or flavor instance variable addressing without using a mapping table).

It is recommended that the following be used during debugging:

```
(proclaim '(optimize (safety 2)))
```

Also, the following should be used before compiling a program one last time after it has been checked out:

```
(proclaim '(optimize (safety 0) (space 2) (compilation-speed 0)))
```

Note that a value of 2 is used in these global declarations to allow another quality to have the higher value of 3 in a local declaration. For example, functions that are frequently called could contain the following:

```
(declare (optimize speed))
```

When you compile using `(safety 0)`, it is best to have the program loaded before recompiling so that all of the definitions are available to the compiler.

Unlike other proclamations, an **optimize** declaration specifier used as an argument of **proclaim** in a file being compiled is effective only during compilation of that file, not when the file is loaded.

declaration {*name*}+ *[c]* Declaration Specifier

If you use nonstandard declarations, you should proclaim *name* globally within this declaration specifier so that Common Lisp compilers that do not understand these nonstandard declarations will ignore them. This form indicates that the *name* declarations are going to be used and prevents the compiler from issuing warnings about these declarations being unrecognized. You can use this declaration specifier only within **proclaim**.

The following declarations are Explorer extensions and are significant only when they apply to an entire **defun**.

arglist . *lambda-list* Declaration Specifier

This declaration specifier records *lambda-list* as the descriptive argument list of the function to be used instead of its real lambda list, if anyone asks what the function's arguments are. This specifier is purely documentation. Note that this syntax line is in the form of a dotted list. It is described in this way only because the meaning of *lambda-list* is already established. Of course, in practice it does not matter if you write a dotted list whose cdr is a list or simply write a canonical list. For example:

```
(defun foo (&rest args)
   (declare (arglist x y &rest z))
      ...)
```

values {*return-value*}* Declaration Specifier

This declaration specifier records *return-values* as the return values list of the function, to be used if anyone asks what values it returns. This specifier is purely documentation. For example:

```
(defun foo ()
   (declare (values w))
      ...)
```

sys:function-parent *parent-function-spec* Declaration Specifier

This declaration specifier records *parent-function-spec* as the parent of this function. If, in the editor, you ask to see the source of this function and the editor does not know where it is, the editor shows you the source code for the parent function instead.

For example, the accessor functions generated by **defstruct** have no **defuns** of their own in the text of the source file. So **defstruct** generates them with **sys:function-parent** declarations, giving the name of the **defstruct** as the parent function spec. When you attempt to edit a definition of an accessor function using META-. the editor positions point at the **defstruct** definition.

:self-flavor *flavor-name* Declaration Specifier

This declaration specifier makes instance variables of the flavor *flavor-name* in self, accessible in the function.

Declarations for Returned Values

13.4 Besides declaring the types of variables with the type and ftype declarations, you can declare the type of an evaluated form's returned value. This kind of declaration can be made using the **the** special form.

the *value-type form* *[c]* Special Form

This special form evaluates *form* and returns its value, which is locally declared to be of type *value-type*. The *value-type* argument is not evaluated. For example:

```
(= 1 (the integer (foo x)))
```

In this example, the compiler is notified that it is safe to use integer comparison rather than allowing for all types of numbers.

You can also use the **values** type specifier with **the** to declare types for multiple returned values. For example:

```
(the (values integer integer) (floor 11 4))
```

The form returning multiple values (in this case `floor`) must return as many values as `values` is expecting. Returning more values is not an error, but the type of the values is unrestricted.

Even if you do not specify *value-type* using the **values** type specifier, it is equivalent to the form (**values** *value-type*) with regard to the rules governing multiple values; that is, if no values are returned—if *form* is equivalent to specifying (**values**)—an error is signaled. If multiple values are returned, then there is no restriction on the type of the second returned value and any subsequent returned values.

If you want the type of an expression to be checked at run time and you want an error reported if it is not what it should be, use **check-type** (described in Section 20, Error Handling).

Global Variables and Named Constants

13.5 The following macros are used for implementing global variables and named constants. These forms establish globally pervasive special declarations for a given variable. By convention, global special variable names begin and end with asterisks.

Note that global variable definitions come in two varieties: **defvar** and **defparameter**. Although both create global special variables, the manner in which they initialize those variables differs in intent and implementation. Specifically, some global variables have their values changed to reflect the current state of the data processing, whereas others remain relatively constant and are in some ways considered parameters to the algorithm. To define these global variables, use the **defvar** and **defparameter** forms, respectively. For example, a variable that reflects the current time should be defined with a **defvar**, whereas a variable that reflects the current time zone should be defined with a **defparameter**.

defvar *variable* &optional *initial-value documentation* *[c]* Macro

This macro is the recommended way to declare the use of a global variable in a program.

Placed at top level in a file, this form declares *variable* globally special and records its location in the file for the sake of the editor so that you can ask to see where the variable is defined. The documentation string is remembered and returned if you invoke `(documentation variable 'variable)`.

If you do not supply an initial value, the variable remains unbound. If you wish to supply a documentation string but no initial value, use the symbol **:unbound** as the *initial-value* form. If *variable* has no value prior to the evaluation of the **defvar**, it is initialized to the result of evaluating the form *initial-value*. The *initial-value* argument is evaluated only if it is to be used. Specifically, note that reloading a file that contains **defvars** does not reinitialize the global variables unless the file is a patch file (see Section 23, Maintaining Large Systems). If you intend for them to be reinitialized, you should probably use **defparameter**.

Using a documentation string has advantages over using a comment to describe the use of the variable because the documentation string is accessible to system programs that can show the documentation to interested users who are using the machine. Although it is still permissible to omit *initial-value* and the documentation string, it is recommended that you put a documentation string in every **defvar**.

The **defvar** macro should be used only at top level, never in function definitions, and only for global variables (those used by more than one function). The form `(defvar foo 'bar "documentation")` is roughly equivalent to the following:

```
(proclaim '(special foo))
(setf (documentation 'foo 'variable) "documentation")
(if (not (variable-boundp foo))
    (setf foo 'bar))
```

If in the editor you mark a region that contains a **defvar** and either compile or evaluate it and if *variable* already has a value, **defvar** does not reassign *variable* to *initial-value*. If *variable* does not have a value, then the assignment is made. If you do not explicitly mark the region but use the default enclosing definition, then the assignment is always made.

defparameter *variable initial-value* &optional *documentation* *[c]* Macro

This macro is the same as **defvar**, except that **defparameter** always sets the variable to the initial value regardless of whether it is already bound. The **defparameter** macro always sets the variable to the specified value so that if, while developing or debugging the program, you change your mind about what the value should be and you then evaluate the **defparameter** form again, the variable receives the new value. It is *not* the intent of **defparameter** to declare that the value of *variable* will never change; for example, **defparameter** does *not* permit the compiler to make assumptions about the value of *variable* in programs being compiled.

As with **defvar**, it is good programming practice to include a documentation string in every **defparameter**.

defconstant *symbol value* &optional *documentation* *[c]* Macro

This macro defines a true constant. The compiler is permitted to assume it will never change. Therefore, if a function that refers to the value of *symbol* is compiled, the compiled function may contain *value* hard coded into it and may or may not actually refer to *symbol* at run time.

The only legal way to change the value of a constant is by reexecuting the **defconstant** with a new *value*. If you change a constant value, it is necessary to recompile any compiled functions that refer to the value of *symbol*.

In a file being compiled, a **defconstant** form is evaluated at compile time for the benefit of possible references later in the file. Consequently, the *value* expression should not reference variables or functions defined earlier in the same file because these values are not known at compile time. However, it is acceptable to use constants and macros.

constantp *object* *[c]* Function

This predicate returns true if *object* is a constant. Constants always evaluate to the same value. Examples of constants are numbers, characters, strings, bit-vectors, keywords, and any symbols defined as constants by **defconstant** (such as **t**, **nil**, and **pi**). Also, a **quote** form is a constant. Consider the following examples:

```
(constantp 5) => true

(constantp 'x) => false

(constantp ''x) => true
```

CONTROL STRUCTURES

14

Introduction

14.1 The following functions are the basic forms for controlling the flow of execution in a Lisp program. These control structures can be classified into three categories: conditional structures, sequential, and iterative.

Conditionals

14.2 The following macros and special forms are conditional control structures.

if *predicate-form then-form* [*else-form*] *[c]* Special Form
if *predicate-form then-form* {*else-form*}* Special Form

This special form is the simplest conditional form. The *predicate-form* argument is evaluated, and if the result is true, the *then-form* is evaluated and its result is returned. Otherwise, the *else-form* is evaluated and its result is returned. The *else-form* defaults to **nil**.

As an Explorer extension, if there are more than three subforms, **if** assumes you want more than one *else-form*; if *test* returns **nil**, they are evaluated sequentially and the result of the last one is returned.

Consider the following example:

```
(defun divide (x y)
   (if (= y 0)
       (print "ERROR - denominator equal to zero.")
       (/ x y)))
```

when *predicate-form* {*body-form*}* *[c]* Macro

If *predicate-form* evaluates to true, the *body-forms* are executed in sequence and the value of the last form is returned. Otherwise, the value of the **when** macro is **nil** and the *body-forms* are not executed.

unless *predicate-form* {*body-form*}* *[c]* Macro

If *predicate-form* evaluates to **nil**, the *body-forms* are executed in sequence and the value of the last form is returned. Otherwise, the value of the **unless** is **nil** and the *body-forms* are not executed.

cond {(*predicate-form* {*body-form*}*)}* *[c]* Special Form

This special form consists of the symbol **cond** followed by several *clauses*. Each clause is a list consisting of a *predicate-form*, called the *condition*, followed by zero or more *body-forms*:

(**cond** (*predicate-form body-form body-form. . .*)
       (*predicate-form*)
       (*predicate-form body-form. . .*)
       . . .)

The idea is that each clause represents a case that is selected if its condition is satisfied and the conditions of all preceding clauses were not satisfied. When a clause is selected, its *body-forms* are evaluated.

The **cond** form processes its clauses in order from left to right. First, the condition of the current clause is evaluated. If the result is **nil, cond** advances to the next clause. Otherwise, the cdr of the clause is treated as a list of *body-forms* that are evaluated in order from left to right. After evaluating the *body-forms*, ~~**cond** returns without inspecting any remaining clauses.~~ The value of the **cond** form equal to any values of the last *body-form* evaluated, or the value of the *predicate-form* if there were no *body-forms* supplied. If the *predicate-form* produces multiple values, only the first value is returned. If **cond** runs out of clauses, that is, if every condition evaluates to **nil** and thus no case is selected, the returned value of the **cond** is **nil**. For example:

```
(cond ((zerop x) (+ y 3))   ; (zerop x) is the predicate and if non-nil,
                            ; (+ y 3) is returned.

      ((null y) (setq y 4)
                (cons x z))
                            ; If y is nil then execute the setq form and
                            ; return the value of the cons form.

      (z)     ; A clause with no body forms. If z is
              ; non-nil then return it.

      ((some-function-returning-multiple-values))
          ; If a predicate returns multiple values, only the first value is
          ; used in the test. If the value is true and there are no consequence
          ; clauses, only the first value of the predicate is returned.

      (t 105))    ; A predicate of t is always satisfied. This is like an "otherwise"
                  ; clause. If the above predicates are not satisfied
                  ; return 105 as a last resort.
```

cond-every {(*predicate-form* {*body-form*}*)}* Macro

This macro has the same syntax as the **cond** macro but executes every clause whose *predicate-form* is satisfied, not just the first. If a *predicate-form* is the symbol **otherwise**, it is satisfied if and only if no preceding *predicate-form* is satisfied. The value returned is the value of the last *body* form in the last clause whose *condition* is satisfied. Multiple values are not returned. For example:

```
(defun foo (x y)
 (cond-every ((zerop x) (setq x 1))
             ((zerop y) (setq y 1))
             ((< x 0) (setq x (abs x)))
             ((< y 0) (setq y (abs y)) (+ x y))
             (otherwise (+ x y))))

(foo 0 0) => 1
(foo -1 -1) => 2
(foo 2 3) => 5
```

case *key-form* {(*test* {*body-form*}*)}* *[c]* Macro

This macro is a conditional that chooses one of its clauses to execute by comparing the value of a form against various constants using **eql**. Its form is as follows:

(case *key-form*
        (*test body. . .*)
        (*test body. . .*)
        (*test body. . .*)
        . . .)

First, **case** evaluates *key-form*. Suppose the resulting value is called *key*. Then, **case** considers each of the clauses in turn. If *key* is **eql** to the clause's *test*, the body of the clause is evaluated and **case** returns the value of the last body form. If there are no matches, **case** returns **nil**.

A *test* can be one of the following:

- A symbol, number, or character object. If the *key* is **eql** to the symbol, number, or character object, it matches.
- A list. If the *key* is **eql** to one of the elements of the list, then it matches. The elements of the list should be symbols, numbers, or character objects.
- The values **t** or **otherwise**; the symbols **t** and **otherwise** are special test identifiers that match anything. Either symbol can be used. To be useful, this clause should be the last in the **case** form.

Note that the *test* arguments are *not* evaluated; if you want them to be evaluated, use **select** rather than **case**. Consider the following example:

```
(case x
  (foo (do-this))
  (bar (do-that))
  ((baz quux mum) (do-the-other-thing))
  (otherwise (ERROR "Never heard of ~S" x)))
```

select *key-form* {(*test* {*body-form*}*)}* Macro

This macro is like **case**, except that the elements of *test* are evaluated before they are used, and testing is done with **eq** instead of **eql**.

Because the *test* items are evaluated, function calls can be used as *test* items. If you make one of your testing forms a function call, then you must use the list syntax to specify your *test* items, even if there is only one *test* item in the clause (to distinguish lists from function calls). For example:

```
(select (scale-of-1-to-10 user-id)
  ;; Simple case of one test item.
  (0
     (print "This person is dead."))
  ;; Case of a list of test items.
  ((1 2 3)
     (print "This person needs help."))
  ;; Case of a single test item that is a function call. Must use the
  ;; list syntax for test items.
  (((ideal-programmer-index :fortran))
     (print "This person could be productive."))
  ;; Case of multiple test items, some of which are function calls.
  (((ideal-programmer-index :lisp)
     (ideal-programmer-index :prolog)
     (ideal-programmer-index :scheme)
     9 10)
      (print "This person has definite prospects."))
  ;; Simple otherwise clause.
  (otherwise
      (funcall acme-referral-service user-id)))
```

selector *key-form comp-fn* {(*test* {*body-form*}*)}* Macro

This macro is like **select**, except that you can specify that the function *comp-fn* is to be used for the comparison instead of eq. For example:

```
(selector (frob x) equal
  ((´(one . two)) (frob-one x))
  ((´(three . four)) (frob-three x))
  (otherwise (frob-any x)))
```

select-match *key-form* {(*pattern condition* {*body-form*}*)}* Macro

This macro is like **select**, but each clause can specify a pattern to compare with the key.

The value of *key-form* is compared with each element of *pattern*, one at a time, until a match is found and the accompanying *condition* evaluates to a non-**nil** value. At this point, the body of that clause is executed and its value is returned. If all the patterns or conditions fail, the body of the **otherwise** clause (if any) is executed. A pattern can test the shape of the key object and can set the variables to which the *condition* form can refer. All the variables set by the patterns are bound locally to the **select-match** form.

The patterns are made with backquotes (see paragraph 18.3.2. Macro Expansion Using the Backquote). Whereas a backquote expression normally indicates how to construct a list structure out of constant and variable parts, in this context it indicates how to match list structure with constants and variables. Constant parts of the backquote expression must match exactly; variables preceded by commas can match anything, but they set the variable to whatever is matched. (Some of the variables may be set even if there is no match.) If a variable appears more than once, it must match the same thing (**equal** list structures) each time. The **ignore** variable name can be used to match anything and ignore it.

For example, `` `(x (,y) . ,z) `` is a pattern that matches a list with at least two elements whose first element is `x` and whose second element is a list of one element. If a list matches, the **caadr** of the list is stored into the value of `y` and the **cddr** of the list is stored into `z`. Consider the following example:

```
(select-match ´(a b c)
   (`(,x b ,x) t (vector x))
   (`((,x ,y) b . ,ignore) t (list x y))
   (`(,x b ,y) (symbolp x) (cons x y))
   (otherwise ´lose-big))
```

This form returns `(a . c)`, having checked `(symbolp ´a)`. The first clause matches only if there are three elements, if the first and third elements are **equal**, and if the second element is `b`. The second clause matches only if the first element is a list of length two and if the second element is `b`. The third clause accepts any list of length three whose second element is `b` and whose car is a symbol. The fourth clause accepts anything that does not match the previous clauses.

The **select-match** macro generates highly optimized code using special instructions.

dispatch *byte-specifier integer* {({*test*} {*body-form*}*)}* Macro

This macro is the same as **select**, but the key is obtained by evaluating (**ldb** *byte-specifier integer*) and the *test*s are all numbers. For example:

```
(princ (dispatch (byte 2 13) cat-type
                 (0 "Siamese.")
                 (1 "Persian.")
                 (2 "Alley.")
                 (3 (error nil "~S is not a known cat type."
                           cat-type))))
```

The arguments *byte-specifier* and *integer* are both evaluated. Byte specifiers and **ldb** are explained in paragraph 3.11, Byte Manipulation Functions.

It is not necessary to include all possible values of the byte that is dispatched on because **dispatch** returns **nil** if no *test* is satisfied.

selectq-every *key-form* {(*test* {*body-form*}*)}* Macro

This macro has the same syntax as **case** and uses the **eql** test, but, like **cond-every**, it tests every clause instead of stopping after satisfying the first one. If an **otherwise** clause is present, it is selected if and only if no preceding clause is selected. The value returned is the value of the last form in the last selected clause. Multiple values are not returned. For example:

```
(selectq-every animal
  ((cat dog) (setq legs 4))
  ((bird man) (setq legs 2)
  ((cat bird) (bad-mix animal))
  ((cat dog man) (beware-of animal)))
```

eval-when ({*situation*}*) {*body-form*}* *[c]* Special Form

This special form is used principally to control when a particular *body-form* should be evaluated during the **make-system** process (or however you initially build the system). The **eval-when** special form is usually a top-level form in your source file and is seldom useful in other places.

The special forms and macros that are commonly used at top level are specially recognized and are processed at an appropriate time. For example, when a file is compiled, a **defun** form does not take effect until the object file is loaded, but **proclaim** and **defmacro** do take effect during compilation because they can affect how the rest of the file is compiled. Occasionally, it may be necessary to specify explicitly when a form is to be processed; this can be done with the special form **eval-when**.

The *situation* argument is not evaluated and contains one or more of the following symbols:

- **eval** — Indicates that when the file is interpreted, the body forms should be executed.

- **compile** — Indicates that when the file is compiled, the compiler should execute the body forms at compile time in the compilation context.

- **load** — Indicates that the compiler should place the body forms in the object file for execution when the file is loaded.

Suppose that you have the following three forms in a file:

```
(eval-when (eval) (print 'evaluating-lisp-source))
(eval-when (compile) (print 'compiling-lisp-source))
(eval-when (load) (print 'loading-lisp-source))
```

The following statements are then true:

- Loading the Lisp version of this file prints the `evaluating-lisp-source` message.
- Compiling the file (generating an object file) prints the `compiling-lisp-source` message.
- Loading the compiled version of this file prints the `loading-lisp-source` message.
- If this file were in a Zmacs buffer and you evaluated the buffer or region, then the `evaluating-lisp-source` message is printed.
- If this file were in a Zmacs buffer and you compiled the buffer or region, then both the `compiling-lisp-source` and `loading-lisp-source` messages are printed.

Sequential Control Structures

14.3 The following forms all execute their *body-forms* sequentially.

progn {*body-form*}* *[c]* Special Form

The *body-forms* are evaluated in sequence from left to right, and the value of the last form is returned. The **progn** special form is the primitive control structure construct for *compound statements*. A **progn** form returns the value or values of the last form. For example:

```
(progn
 (some-form)
 (another-form)
 ...
 (last-form))    ; The value of this form is returned as the value of the progn.
```

Lambda expressions, **cond** forms, **do** forms, and many other control structure forms use **progn** implicitly; that is, they allow multiple forms in their bodies.

prog1 *first-form* {*additional-form*}* *[c]* Macro

This macro is similar to **progn**, but it returns the value of its *first-form* rather than its last. It is most commonly used to evaluate an expression with side effects and to return a value that must be computed *before* the side effects happen. For example:

```
(setq x (prog1 y
               (setq y x)))
```

This form interchanges the values of the variables `x` and `y`. The **prog1** macro never returns multiple values (see **multiple-value-prog1** in paragraph 16.5, Passing and Receiving Multiple Values).

prog2 *first-form second-form* {*additional-form*}* *[c]* Macro

This special form is similar to **progn** and **prog1**, but it returns the value of its *second-form*.

The previously discussed **prog**-style forms have two aspects in common: they all have a block of code that is executed in sequence, and they all return values implicitly. The **block** form is similar, but it allows you to incorporate an explicit **return** form and also gives a name to the block of code to be executed. Described after the **block** special form are the different explicit return forms available.

block *name* {*body-form*}* *[c]* Special Form

This special form executes the body, returning the values of the last *body-form*, but permitting nonlocal exit using **return-from** forms present lexically within the body. The *name* argument is a symbol that is not evaluated and is used to match **return-from** forms with their **blocks**. For example:

```
(block foo
  (return-from foo 24) t) => 24
(block foo t) => t
```

return-from *name* &optional *value* *[c]* Special Form

This special form performs a nonlocal exit from the lexically innermost **block** whose name is *name*. The argument *name* is not evaluated. When the compiler is used, the **return-from** forms are matched with **block** forms at compile time. Functions defined by **defun** have an implicit **block** that surrounds their bodies if the function name is a symbol rather than a list; therefore, when this form is used in a **defun**, the *name* argument can be the function's name.

The *value* argument is evaluated and its value or values are returned as the value of the exited **block** form. If *value* is not supplied, it defaults to **nil**.

A **return-from** form can appear as or inside an argument to a regular function, but if the **return-from** is executed, then the original function is never actually called. Consider the following example:

```
(block done
    (foo (if x (return-from done t) nil)))
```

The function `foo` is actually called only if the value of `x` is `nil`. Of course, if `foo` were a macro or special form that did not evaluate the `return-from` as an argument, then `foo` might be called. This style of programming can be confusing and is not recommended.

return &optional *values* *[c]* Special Form

This special form is equivalent to the following form:

(**return-from nil** *values*)

It returns from a **block** whose name is **nil**; such blocks are implicitly created by forms such as **prog**, **do**, and **loop**.

comment *{form}** Macro

This macro is used to allow any number of forms to be ignored. The symbol **comment** is returned. It is most useful for commenting out function definitions that are not needed but are worth preserving in a source file. See also the Reader macro #| in the Input section of the *Explorer Input/Output Reference* manual.

The main difference between the **comment** macro and the Reader macro is that the Reader macro ignores the commented items more completely, such as other Reader macros and references to symbols in nonexistent packages. Since the **comment** macro is initially processed by the Reader macro, all the referenced symbols are found by **intern**. Therefore, make sure that all the referenced packages exist.

Iterative Control Structures

14.4 Iteration is performed by looping constructs, mapping constructs, or various prog-related constructs.

Looping Constructs

14.4.1 The following forms are associated with looping control structures.

loop *{body-form}** *[c]* Macro

In Common Lisp, this macro is used to execute the *body-forms* until an exit form (such as **return** or **return-from**) has been encountered; if one is not encountered, this construct is an infinite loop. The Explorer system also supports a more advanced **loop** facility in which atoms appearing in the body have special meaning for controlling the loop; this facility is explained in Section 15, Loop Iteration Macro. The advanced **loop** macro is used if the first *body-form* is atomic.

do ({*var* |(*var* [*init* [*step*]])}*) (*end-test* {*result-form*}*)
{*declaration*}* {*tag* | *statement*}* *[c]* Macro

do* ({*var* |(*var* [*init* [*step*]])}*) (*end-test* {*result-form*}*)
{*declaration*}* {*tag* | *statement*}* *[c]* Macro

The **do** macro provides a simple, general iteration facility with an arbitrary number of iteration variables whose values are saved when the **do** is entered and are restored when it is exited. The iteration variables are used in the iteration performed by **do**. At the beginning, they are initialized to specified values; at the end of each cycle around the loop, the values of the iteration variables are changed according to step rules. The **do** macro allows the programmer to specify a predicate that determines when the iteration terminates and the value to be returned as the result of the **do** form.

A typical **do** form looks like this:

```
(do ((var1 init1 step1)
     (var2 init2 step2) . . .)
    (end-test result-form1 result-form2 . . .)
  {declaration}*
    body . . .)
```

The first item in the form is a list of zero or more iteration variable specifiers. Each specifier is a list of the name of a variable *var*, an initial value form *init* (which defaults to **nil** if it is omitted), and an update form *step*. If the *step* form is omitted, the *var* is not automatically changed between iterations.

An iteration variable specifier can also be simply the name of a variable, rather than a list, in which case it has an initial value of **nil** and is not automatically changed between iterations.

The difference between the **do** and **do*** macros is that for the **do** macro all assignments to the iteration variables are done in parallel. For the **do*** macro, the initializations and updates are done in sequence, so one iteration variable can use the previous variable assignment as part of its calculation.

The second element of the **do** form is a list containing a termination predicate form *end-test* and zero or more *result-forms*. This element resembles a **cond** clause. At the beginning of each iteration and after processing of the variable specifiers, the *end-test* is evaluated. If the result is **nil**, execution proceeds with the body of the **do**. If the result is true, the *result-forms* are evaluated from left to right, and then **do** returns. The value of the **do** is the value of the last *result-form* or **nil** if there were no *result-forms* (not the value of the *end-test*, as you might expect by analogy with **cond**).

Note that the *end-test* is evaluated before the first time the body is evaluated. The **do** macro first initializes the variables from the *init* forms; then it checks the *end-test*, processes the body, deals with the *step* forms, tests the *end-test* again, and so on. If the *end-test* returns a non-**nil** value the first time, then the body is not executed.

If the *end-test* is **nil**, then it is never true and there are no exit *result-forms*. Therefore, the body of the **do** is executed repeatedly, making it analogous to the **do-forever** form. An infinite loop of this kind can be terminated by use of **return** or **throw**.

The **do** macro implicitly creates a **block** with name **nil**, so **return** can be used lexically within a **do** to exit it immediately. The **do** form returns whatever values were specified in the **return** form. See paragraph 14.3, Sequential Control Structures, for more information. The body of the **do** is actually treated as a **tagbody** so that it can contain **go** tags (see paragraph 14.4.3, Other Iterative Control Structures), but this usage is discouraged because it is often unclear.

Consider the following examples:

```
;; This do sets every element of foo-array to 0.
(do  ((i 0 (1+ i))  ; “i” is initialized to 0 and incremented by 1 with each iteration.
                    ; “n” is initialized to the length of the array and
                    ; does not change with each  iteration.
     (n (length foo-array)))

                    ; This is the exit test. When true, it returns nil
                    ; for the value of the do.
     ((= i n))

                    ; This is the body form of the do.
     (setf (aref foo-array i) 0)

(do ((z lst (cdr z))             ; z starts as lst and is cdr'd each time.
    (y other-lst)                ; y starts as other-lst and is unchanged.
     (x)                         ; x starts as nil and is unchanged here.
     w)                          ; w starts as nil and is unchanged here.
     (nil)                       ; end test is nil so this is an infinite loop
     ...)                        ; if there is no return statement in the body.
```

The body of a **do** may contain no forms at all. Very often an iterative algorithm can be most clearly expressed entirely in the *steps* and *forms* of the **do**, and the body is empty. For example:

```
(do ((x x (cdr x))
     (y y (cdr y))
     (z nil (cons (f x y) z)))  ; Uses parallel assignment.
   ((or (endp x) (endp y))
     (nreverse z))              ; Typical use of nreverse.
  )                             ; No body required.
```

The **do*** macro is similar to **do** but initializes and updates its variables sequentially rather than in parallel.

dolist (*var listform* [*result-form*]) {*declaration*}* {*body-form*}* *[c]* Macro

This macro is a convenient abbreviation for the most common list iteration. This macro performs *body-form* once for each element in the list that is the value of *listform*, with *var* bound to the successive elements. When the list is exhausted, the value of *result-form* is returned; **nil** is returned if *result-form* is missing. The *body-form* of **dolist** allows **tagbody** tags, **go**, and **return** statements. For example:

```
(dolist (item (frobs foo))
   (mung item))
```

The preceding form is equivalent to the following:

```
(do ((lst (frobs foo) (cdr lst))
     (item))
    ((endp lst))
  (setq item (car lst))
  (mung item))
```

Note that the **dolist** example is much simpler than the **do** example and does not use the variable `lst`.

dotimes (*index count* [*result-form*]) {*declaration*}* {*body-form*}* *[c]* Macro

This macro is a convenient abbreviation for the most common integer iteration. This macro performs *body-form* the number of times given by the value of *count*, with *index* bound to 0, 1, and so on up to one less than *count*. When *index* has reached *count*, the value of *result-form* is returned; **nil** is returned if *result-form* is missing. For example:

```
(dotimes (i 10 (print x))
   (setf x (+ i 2))) => 11
```

Mapping

14.4.2 *Mapping* is a kind of iteration in which a specified function is successively applied to pieces of a list (however, **map**, described in Section 9, Sequences, works on sequences of all kinds).

mapcar *fn* {*list*}* *[c]* Function
maplist *fn* {*list*}* *[c]* Function
mapc *fn* {*list*}* *[c]* Function
mapl *fn* {*list*}* *[c]* Function
mapcan *fn* {*list*}* *[c]* Function
mapcon *fn* {*list*}* *[c]* Function

These functions successively apply *fn* to pieces of each *list*. How the pieces are chosen depends on the function used.

For example, **mapcar** operates on successive elements of the list. As it goes down the list, it calls the function, giving it an element of the list as its one argument: first the car, then the cadr, then the caddr, and so on, continuing until the end of the list is reached. The value returned by **mapcar** is a list of the results of the successive calls to the function. For example, you could use **mapcar** to call the function **abs** for each element of the list in the following example:

```
(mapcar #'abs '(1 -2 -4.5 6.0e15 -4.2))
=> (1 2 4.5 6.0e15 4.2)
```

In general, the mapping functions take any number of arguments. For example:

(**mapcar** *fn x1 x2* . . . *xn*)

In this case, *fn* must be a function of *n* arguments. The **mapcar** function proceeds down the lists *x1, x2, ..., xn* in parallel. The first argument to *fn* comes from *x1*, the second from *x2*, and so on. The iteration stops as soon as one of the lists is exhausted. If you want to call a function of many arguments where one of the arguments successively takes on the values of the elements of a list and the other arguments are constant, you can use a circular list for the other arguments to **mapcar**. The function **circular-list** is useful for creating such lists; see paragraph 6.4, Functions Associated With Lists.

There are five other mapping functions besides **mapcar**. The **maplist** function is like **mapcar**, except that the function is applied to the list and to subsequent cdr's of that list rather than to subsequent elements of the list. The **mapc** and **mapl** functions are like **mapcar** and **maplist**, respectively, except that they return the second argument (the first sequence). These functions are used when the function is being called merely for its side effects rather than for its returned values. The **mapcan** and **mapcon** functions are like **mapcar** and **maplist**, respectively, except that they combine the results of the function using **nconc** instead of **append**. That is, **mapcon** could have been defined by the following form:

```
(defun mapcon (f x y)
   (apply #'nconc (maplist f x y)))
```

Sometimes a **do** or a straightforward recursion is preferable to a map; however, the mapping functions should be used wherever they naturally apply because this increases the clarity of the code.

Often *fn* is a lambda expression rather than a symbol; for example:

```
(mapcar #'(lambda (x)
            (cons x something))
  some-list)
```

The functional argument to a mapping function must be a function acceptable to **apply**; it cannot be a macro or the name of a special form.

The following table shows the relations between the six map functions.

Returns:	**Applies Function To:**	
	Successive Sublists	**Successive Elements**
Its own second argument	**mapl**	**mapc**
list of the function results	**maplist**	**mapcar**
nconc of the function results	**mapcon**	**mapcan**

Other Iterative Control Structures

14.4.3 Iteration can be performed with the various special forms below. However, these special forms are not recommended because they tend to encourage unstructured programming.

prog ({*var* | (*var* [*init*])}*) {*declaration*}* {*tag* | *body-form*}* *[c]* Special Form
prog* ({*var* | (*var* [*init*])}*) {*declaration*}* {*tag* | *body-form*}* *[c]* Special Form

The **prog** form actually performs three distinct operations: it binds *var* local variables to *init* forms, permits use of a **return** statement, and permits use of a **go** form. In Common Lisp, these three operations are separated into three distinct constructs: the **let**, **block**, and **tagbody** constructs. Now **prog** is obsolete, because it is much cleaner to use **let**, **block**, **tagbody**, or all three of them. However, **prog** continues to be supported because it is used so extensively in old programs.

These three constructs can be used independently as building blocks for other types of constructs.

See also the **do** special form (paragraph 14.4.1, Looping Constructs), which uses a body similar to **prog**. The **do**, **catch**, and **throw** special forms are included in Common Lisp as an attempt to encourage a programming style devoid of goto forms. This style often leads to more readable, more easily maintained code. It is recommended that the programmer use these forms instead of **prog** wherever reasonable.

CAUTION: The prog macro does not return as its value the last form in its body. It returns nil unless an explicit return statement is used.

A typical **prog** looks like (**prog** (*variables* ...) *body*...) and is equivalent to the following:

```
(block nil
   (let (variables ...)
      (tagbody body ...)))
```

The **prog*** special form is almost the same as **prog**, except that the binding and initialization of the variables in **prog*** is done sequentially to allow variables to be initialized to their predecessors' *init* form. Thus, it uses **let*** instead of **let**.

tagbody {*tag* | *body-form*}* *[c]* Special Form

This special form executes all *body-forms*, which are lists, and returns **nil**. All *tags*, which are symbols, are available for use with the **go** statement. Any comparison of *tag* names is performed using **eql**.

CAUTION: The tagbody special form does not return as its value the last form in its body. It returns nil unless an explicit return statement is used.

A *tag* name can appear only once within the **tagbody**, although the same *tag* name can be used within a nested **tagbody**, in which case the outer name is shadowed. Within the **tagbody**, anything other than a symbol, integer, or list produces an error.

go *tag* *[c]* Special Form

This special form is used to branch to a *tag* defined in a lexically containing **tagbody** form (or other forms that implicitly expand into a **tagbody**, such as **prog**, **do**, or **loop**). The argument *tag* must be a symbol. It is not evaluated.

Dynamic Nonlocal Exits

14.5 A *dynamic nonlocal exit* allows you to exit a computation, in practice always a stack group, and resume execution at another point within the current dynamic scope. In Common Lisp, dynamic nonlocal exits are performed by *catch* and *throw* forms. The **catch** form executes its subforms like **progn** unless a **throw** form (which can be outside the lexical scope of the catch) is executed, in which case the catch ceases execution and returns a value or values indicated by the throw.

catch *tag* {*body-form*}* *[c]* Special Form

This special form is used with the **throw** form to perform nonlocal exits. First, *tag* is evaluated; the result is called the tag of the **catch**. Then the *body-forms* are evaluated sequentially, and the value of the last form is returned, including multiple values, if any. However, if, during the evaluation of the body, the function **throw** is called with the same tag as the tag of the **catch**, then the evaluation of the body is discontinued, and the **catch** form immediately returns the values of the second argument to **throw** without further evaluating the current *body-form* or the rest of the body.

The *tag* forms are used to match **catches** with **throws**. The following form catches a `(throw 'foo nil)` form:

```
(catch 'foo . . .)
```

However, it does not catch the following form:

```
(throw 'bar nil)
```

An error is signaled if **throw** is invoked when there is no suitable **catch** (or **catch-all**, which is explained later).

The values **t** and **nil** for *tag* are special. These values are only for internal use by **unwind-protect** and **catch-all**, respectively.

Consider the following example:

```
;; Give infinity a value.
(defconstant infinity most-positive-long-float)

(defun zerodiv (x y)  ; Allow zero division.
 (catch 'yes-it-is
           (/ x (isit-0? y))))

(defun isit-0? (z)    ; Return infinity if z is zero.
  (if (zerop z)
      (throw 'yes-it-is infinity)
       z))

 (zerodiv 1 0) => 8.98846567e307

 (zerodiv 1.0 2.0) => 0.5
```

This simple example shows how the **catch** and **throw** forms operate. In the function `zerodiv`, a `catch` is established at the beginning of the function. Thereafter, if a `throw` whose tag matches the `catch` is executed during the processing of this function, the returned value of the `throw` is returned as the value of the `catch`. This occurs when the form `(zerodiv 1 0)` is executed. If a `throw` form is not executed during the processing of the `catch`, then the value of the last subform of the `catch` is returned. This occurs when the form `(zerodiv 1.0 2.0)` is executed.

The next example shows the dynamic scope of **catches** and **throws**:

```
(defun funa (x)
  (catch 'trap
         (+ 3 (funb x))))

(defun funb (y)
  (catch 'trap
         (- 2 (func y))))

(defun func (z)
   (if (minusp z)
      (throw 'trap z)
      z))

(funa -3) => 0

(funa 3) => 2
```

In this example, when the form `(funa -3)` is evaluated, a `throw` in `func` is encountered, but there are two tags with the name `trap`: one in `funa` and one in `funb`. Because the tags are dynamically scoped, the value of `z` is returned to the `catch` in `funb` instead of `funa`. (Dynamic scope implies that when two objects of the same name are being referenced, the most recently established of the two is the one to be referenced.) When the form `(funa 3)` is invoked, a throw is not encountered; therefore, all three functions are completely evaluated.

Note what happens when the tag form in `funb` is changed:

```
(defun funa (x)
  (catch 'trap (+ 3 (funb x))))

(defun funb (y)
  (catch 'snare (- 2 (func y))))

(defun func (z)
  (if (minusp z)
      (throw 'trap z)
       z)

(funa -3) => -3
```

In this example, no shadowing occurs, the value of the `throw` form is returned to the `catch` form in `funa`, and the subform of `funa` is not processed.

unwind-protect *protected-form* {*cleanup-form*}* *[c]* Special Form

This special form protects against nonlocal exits by ensuring that the *cleanup-forms* are executed if a nonlocal exit occurs in the *protected-form*. Consider the following example:

```
(progn
  (turn-on-water-faucet)
  (hairy-function 1 2 3)
  (turn-off-water-faucet))
```

The nonlocal exit facility can create a situation in which this code does not work. For instance, if `hairy-function` performs a **throw** to a **catch**, if the user presses ABORT, or if an error that is outside of the `progn` form is signaled, then `(turn-off-water-faucet)` is never evaluated (and the water faucet is presumably left running).

This example can be rewritten as follows:

```
(unwind-protect
  (progn (turn-on-water-faucet)
         (hairy-function 1 2 3))
  (turn-off-water-faucet))
```

If `hairy-function` performs a **throw** to a **catch** that is outside of the `progn` form, then the `(turn-off-water-faucet)` form is executed before the value of the **throw** is returned. If the `progn` form returns normally, then the `(turn-off-water-faucet)` form is evaluated, and the `unwind-protect` returns the result of the `progn`.

Thus, you can use **unwind-protect** to make certain that the *cleanup-forms* at least begin to execute. However, if one of the *cleanup-forms* throws out of the **progn**, then the remaining *cleanup-forms* will not be executed. In either case, the value returned by **unwind-protect** is the result of evaluating *protected-form*; the result of evaluating the *cleanup-forms* is not returned. Furthermore, **unwind-protect** not only guards against a premature exit from *protected-form* via a **throw**, but also via a **go** or **return-from** special form. It also guards against error signals.

catch-continuation *tag throw-cont non-throw-cont* {*body-form*}* Macro
catch-continuation-if *cond-form tag throw-cont non-throw-cont* {*body-form*}* Macro

This macro makes it convenient to pass back multiple values from the body but still indicates whether the exit is normal or due to a throw.

The *body-forms* are executed inside a **catch** on *tag* (which is evaluated). If the *body-forms* return normally, the function *non-throw-cont* is called, with all the values returned by the last form in the *body-form* as arguments. This function's values are returned from the **catch-continuation**.

If, on the other hand, a throw to *tag* occurs, the values it returns are passed to the function *throw-cont*, and its values are returned.

If either of the continuations is explicitly written as **nil**, it is not called at all. The arguments that would have been passed to it are returned instead. This is equivalent to using **values** as the function; but a continuation explicitly written as **nil** is optimized, so use it instead.

The **catch-continuation-if** macro differs only in that the catch is not executed if the value of the *cond-form* is **nil** when the **catch-continuation-if** is entered (not when the throw occurs). In this case, the non-throw continuation, if any, is always called.

In the general case, consing is necessary to record the multiple values, but if either continuation is an explicit `#'(lambda ...)` with a fixed number of arguments or if a continuation is **nil**, it is open-coded and the consing may be avoided.

catch-all {*body-form*}* Macro

The form (**catch-all** *body-form*) is like (**catch** *some-tag body-form*), except that it catches a **throw** to any tag at all. The one thing that **catch-all** does not catch is an ***unwind-stack** with a tag of **t**. The **catch-all** macro expands into **catch** with a *tag* of **nil**.

The **catch-all** macro returns all the values thrown to it, or returned by the body. This is a fairly dangerous form; you should use **unwind-protect** instead.

throw *tag values-form* *[c]* Special Form

This special form is the primitive for exiting from a **catch**. The *tag* argument is evaluated, and the result is matched (using **eq**) against the tags of all active **catches**; the innermost matching tag is exited. If no matching **catch** is active, an error is signaled.

All the values of *values-form* are returned from the exited **catch**.

Any **catch** whose tag is **nil** always matches any **throw**. They are really equivalent to **catch-all**, which should be used instead. If the only matching **catches** are **unwind-protects**, then an error is signaled because an **unwind-protect** always throws again after its cleanup forms are finished; if there is nothing to catch after the last **unwind-protect**, an error happens then, and it is better to detect the error sooner.

The values **t** and **nil** for *tag* are reserved and used for internal purposes. The value **nil** cannot be used, because it causes confusion in handling **unwind-protect**. The value **t** can only be used with ***unwind-stack**.

See the description of **catch** earlier in this section for further details.

*unwind-stack *tag value frame-count action* Function

This function is related to **throw** and is provided for stack-manipulating programs such as the debugger. This newest version of ***unwind-stack** is a subset of the ***unwind-stack** of Release 2; it is no longer a generalized **throw**. The arguments for ***unwind-stack** are as follows:

tag — Exists only for partial compatibility with the earlier version of ***unwind-stack**; this argument must be **t**.

value — Can be any Lisp object.

frame-count — Must be a fixnum or **nil**. Either this or the *action* argument must be non-**nil**.

action — Must be a functional object or **nil**. Either this or the *frame-count* argument must be non-**nil**.

The ***unwind-stack** function unwinds the frames on the stack, performing all the clean-up it would perform if it were throwing through them, including the *cleanup-forms* of **unwind-protects**. There are three possible situations:

- If *frame-count* is a fixnum, it specifies the number of frames to unwind. If *action* is **nil**, *value* is returned from the last frame unwound. (The definition of *frame* is necessarily implementation dependent, but in general, each function call creates a frame. The notion of *open* and *active* frames does not exist in Explorer Release 3.)

- If *frame-count* is **nil**, it indicates that all frames in the stack should be unwound. The *action* argument must then be non-**nil** and is called with one argument, *value*, after all the frames have been unwound. The *action* is not permitted to return in this situation. It is often useful for *action* to be a stack group.

- If both *frame-count* and *action* are non-**nil**, then *action* is called with *value* as its argument after *frame-count* frames have been unwound. The *action* form may return, and its values are returned as if from the last frame unwound.

Equality Predicates

14.6 The Common Lisp equality predicates can be ranked from the most specific to the most general: **eq**, **eql**, **equal**, and **equalp**. Any two objects that return true when compared by one equality predicate will also return true when compared by any more general equality predicate.

eq *x y*

[c] Function

This form is true if and only if *x* and *y* are the same object. Being the same object means being located in the same memory location. It should be noted that things that print the same are not necessarily **eq** to each other. For instance, numbers with the same value need not be **eq**, and two similar lists are usually not **eq**. Consider the following examples:

```
(eq 'p 'b) => nil
(eq 'p 'p) => true
(eq (cons 'p 'b) (cons 'p 'b)) => nil
(setf x (cons 'p 'b))
(eq x x) => true
(eq 2 2.0) => nil
(eq 2.0 2.0) => nil
```

On the Explorer system, **eq** works for comparing fixnums and characters (because they are represented as immediate values and not as pointers to a memory location), but you should not rely on this comparison because it may not work on other Lisp implementations. Equality does not imply eqness for other types of numbers. To compare numbers, use the function **=**.

neq *x y*

Function

This predicate is the complement of **eq**:

(**neq** *x y*) <=> (**not** (**eq** *x y*))

This function is provided simply as an abbreviation.

eql *x y*

[c] Function

This predicate is the same as the predicate **eq**, except that if *x* and *y* are numbers or characters, they are **eql** if they are of the same type and if they are numerically equal or represent the same character (for example, a floating-point number is never **equal** to an integer even if the predicate **=** is true of them). Consider the following examples:

```
(eql 'p 'b) => nil
(eql 'p 'p) => true
(eql (cons 'p 'b) (cons 'p 'b)) => nil
(setf x (cons 'p 'b))
(eql x x) => true
(eql 2 2) => true
(eql 2 2.0) => nil
(eql #c(5 -3) #c(5 -3)) => true
```

equal *x y* *[c]* Function

This predicate returns true if its arguments are similar (isomorphic) objects. Two numbers are equal if they have the same value and type. For conses, **equal** is defined recursively as the two cars being **equal** and the two cdrs being **equal**. Two strings are **equal** if they have the same length and if the characters composing them are the same; see the **string=** predicate. Character case is significant in comparisons using this function. All other objects are **equal** if and only if they are **eq**. For example:

```
(equal 'p 'b) => nil
(equal 'p 'p) => true
(setf p '(1 2 3))
(setf b '(1 2 3))
(eq p b) => nil
(equal p b) => true
(equal 2 2) => true
(equal 2 2.0) => nil
(equal 34.0 34.0) => true
(equal #c(5 -3) #c(5 -3)) => true
(equal #c(5 -3.0) #c(5 -3)) => nil
(equal "P" "p") = > nil
```

CAUTION: Care should be taken when applying this predicate to circular list structure because the computations can result in an infinite loop.

Additionally, **eq** always implies **equal**; that is, if `(eq p b)` is true, then so is `(equal p b)`. A rough definition of **equal** is that two objects are **equal** if they look the same when printed out.

To compare a tree of conses using **eql** (or any other desired predicate) on the leaves, use **tree-equal**.

equalp *x y* *[c]* Function

This predicate is a broader kind of equality than **equal**. Two objects that are **equal** are always **equalp**. Additionally, numbers of different types are **equalp** if they are **=**. Two character objects are **equalp** if they are **char-equal** (that is, they are compared ignoring font, case, and modifying bits).

Two arrays of any sort are **equalp** if they have the same dimensions and if corresponding elements are **equalp**. In particular, this means that two strings are **equalp** if they match, ignoring case and font information (that is, according to **string-equal**). For example:

```
(equalp 'p 'b) => nil
(equalp 'p 'p) => true
(equalp 2 2) => true
(equalp 2 2.0) => true
(equalp 2.0 2.0) => true
(equalp #c(26 -34) #c(26 -34)) => true
(equalp #c(26 -34.0) #c(26 -34)) => true
(equalp '(1 "P") '(1.0 "p")) => true
(equalp "P" "p") => true
(equalp #(1 "P") #(1.0 "P")) => true
```

Logical Operators

14.7 The following are the Common Lisp Boolean logical operators, which can be used by themselves or within other constructs to control the flow of execution.

not *x*

[c] Function

This function returns **t** if *x* is **nil**; otherwise, it returns **nil**. The **null** predicate is the same as **not**, but **null** is normally used to test for the end of a list, whereas **not** is used to invert the sense of a logical value. Some people prefer to distinguish between **nil** as falsehood and **nil** as the empty list by using the first of the following forms rather than the second:

```
(cond ((not (null lst)) ... )
      ( ... ))

(cond (lst ... )
      ( ... ))
```

There is no loss of efficiency with either form, because these compile into exactly the same instructions.

and *{form}**

[c] Special Form

This special form evaluates the *forms* one at a time, from left to right. If any *form* evaluates to **nil**, **and** immediately returns **nil** without evaluating the remaining *forms*. If all the *forms* evaluate to non-nil values, **and** returns the value of the last *form*. The **and** special form returns multiple values only if it is the result of evaluating the last *form* in the sequence.

The **and** special form can be used in two different ways. You can use it as a logical **and** function, because it returns a true value only if all of its arguments are true. For example:

```
(when (and socrates-is-a-person all-people-are-mortal)
   (setf socrates-is-mortal t))
```

Because the order of evaluation is well-defined, you can evaluate the following form knowing that the `x` in the `eq` form will not be evaluated if `x` is found to be unbound.

```
(if (and (variable-boundp x) (eq x 'foo))
    (setf y 'bar))
```

You can also use **and** as a simple conditional form:

```
(and (setf temp (assoc x y))
     (setf (cdr temp) z))

(and bright-day
     glorious-day
   (princ "It is a bright and glorious day."))
```

However, **when** is usually preferable in these cases.

Note that `(and) => t`, which is the identity for the **and** operation.

If a form returns multiple values, only the first value is used for the continuation test; if the first returned value is **nil**, the **and** special form returns **nil**. If the first value is true, then the next form is evaluated, unless there is no next form, in which case all of the multiple values are returned.

or *{form}** *[c]* Special Form

This special form evaluates its arguments one at a time from left to right. If none of its argument forms return a non-**nil** value, then **or** returns **nil**. When the first non-**nil** returning argument is encountered, **or** returns that value and ignores the rest of the arguments.

If *form* returns multiple values, only the first value is used for the continuation test; if the first value is true, then multiple values are returned only if this is the last form in the **or** expression. For example:

```
(or (values 1 2) t) => 1
(or nil (values 1 2)) => 1  2
```

As with **and**, **or** can be used either as a logical operator or as a conditional. For example:

```
(or it-is-fish it-is-fowl)

(or it-is-fish it-is-fowl
  (print "It is neither fish nor fowl."))
```

However, you can use **unless** in the latter case, and it is clearer than or. Consider the following two forms:

```
(setf x (if a
            a
            b))

(setf x (or a b))
```

These forms are roughly equivalent except that the `or` form evaluates `a` only once.

Note that `(or) => nil` is the identity for this operation.

xor *{form}** *[c]* Function

This function returns **t** if an odd number of *forms* evaluate to a non-**nil** value; otherwise, it returns **nil**. Note that all *forms* are evaluated.

15

LOOP ITERATION MACRO

Introduction

15.1 The Explorer system has two **loop** constructs, which are two support routines. Section 14, Control Structures, describes the Common Lisp **loop** macro. This section describes an alternate looping construct, which unfortunately has the same name for historical reasons.

The main benefit of using this alternate **loop** is that the syntax provides a very English-like coding style. Because of this and because of the sparse use of parentheses, this alternate **loop** has become popular and is therefore supported in the Explorer system. It should be kept in mind that this alternate **loop** is not part of the Common Lisp standard, nor is it possible to use any of the clauses defined in this section within a Common Lisp **loop**.

The system software distinguishes between the two constructs by looking at the first argument. If it is a list, then the **loop** is of the Common Lisp variety; otherwise, it is this alternate **loop**.

Extended Loop Iteration Facility Description

15.2 Generally, this **loop** macro generates a single program loop into which a large variety of features can be incorporated. The loop consists of some initialization (*prologue*) code, a body that may be executed several times, and some exit (*epilogue*) code. Variables can be declared local to the loop. The features are concerned with loop variables, deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The **loop** form consists of a series of clauses. Within each part of the template filled in by **loop**, these clauses are executed strictly in the order implied by the original composition. Thus, just as in ordinary Lisp code, side effects can be used, and one piece of code may need to follow another to operate properly.

The following meta-description identifies each of the clauses that can appear within the extended **loop** facility.

loop *[***named name***]*

Prologue Clauses:

nodeclare ({*variable*}*)
with *var* = *expr* {**and** *var* = *expr*}*
with ({*var*}*) [*data-type*]
initially *expr*

Iteration Clauses:

for *var* [*data-type*] **in** *expr1* [**by** *expr2*]
for *var* [*data-type*] **on** *expr1* [**by** *expr2*]
for *var* [*data-type*] = *expr1* [**then** *expr2*]
for *var* [*data-type*] **first** *expr1* **then** *expr2*
for *var* [*data-type*] **from** *expr1* [**to** | **downto** | **below** | **above** *expr2*]
[**by** *expr3*]
for *var* [*data-type*] **being** {**each** | **the**} *path*
repeat *count-expr*

Body Clauses:

always *bool-expr*
append *expr* [**into** *var* [*data-type*]]
collect *expr* [**into** *var*]
count *expr* [**into** *var* [*data-type*]]
do *expr*
if *bool-expr true-consequence-clause* [**else** *false-consequence-clause*]
when *bool-expr true-consequence-clause* [**else** *false-consequence-clause*]
maximize *expr* [**into** *var*]
minimize *expr* [**into** *var*]
nconc *expr* [**into** *var*]
never *bool-expr*
return *expr*
sum *expr* [**into** *var*]
thereis *expr*
unless *bool-expr true-consequence-clause*
until *bool-expr*
while *bool-expr*

Epilogue Clause:

finally *expr*

Each of these clauses is discussed in greater detail later in this section.

Note that **loop** forms are intended to look like stylized English rather than Lisp code. Thus, these forms use fewer parentheses, and many of the keywords are accepted in several synonymous forms to allow writing code in almost grammatical English (for example, **do**, **doing**, **collect**, **collecting**, **for**, **as**, and so on). Some programmers find this notation verbose and distasteful, while others find it flexible and convenient. The former are invited to continue to use the Common Lisp **do**.

The following form illustrates the use of **loop**:

```
(defun print-elements-of-list (list-of-elements)
   (loop for element in list-of-elements
         do (print element)))
```

This form prints each element in its argument, which should be a list, and returns **nil**.

The following function `gather-alist-entries` takes an association list and returns a list of the keys; that is:

```
(defun gather-alist-entries (list-of-pairs)
   (loop for pair in list-of-pairs
         collect (car pair)))

(gather-alist-entries '((foo 1 2) (bar 259) (baz)))
=> (foo bar baz)
```

The following function takes two arguments, which should be integers, and returns a list of all the numbers in that range (inclusive) that satisfy the predicate `interesting-p`:

```
(defun extract-interesting-numbers (start-value end-value)
  (loop for number from start-value to end-value
        when (interesting-p number) collect number))
```

In the following example, the function `find-maximum-element` returns the maximum value from the elements of its argument, a one-dimensional array:

```
(defun find-maximum-element (an-array)
  (loop for i from 0 below (length an-array)
        maximize (aref an-array i)))
```

The following function definition, `my-remove`, is like the function **delete**, except that it copies the list rather than destructively splicing out elements. This is similar, although not identical, to the function **remove**:

```
(defun my-remove (object list)
  (loop for element in list
        unless (equal object element) collect element))
```

The following code returns the first element of its list argument that satisfies the predicate `frobp`. If none is found, an error is signaled:

```
(defun find-frob (list)
  (loop for element in list
        when (frobp element) return element
        finally (ferror nil "No frob found in the list ~S" list)))
```

Loop Clauses

15.3 Internally, **loop** creates a **prog** that includes variable bindings, preiteration (initialization) code, post-iteration (exit) code, the body of the iteration, and stepping of variables of iteration to their next values (which happens on every iteration after executing the body).

In the context of this **loop** macro, a *clause* is defined as a series of words or Lisp forms that begin with one of the reserved words listed in paragraph 15.2, Extended Loop Iteration Facility Description. The words **for** and **do** introduce clauses in the following example:

```
(loop for x in l
      do (print x))
```

This form contains two clauses, `for x in l` and `do (print x)`. Certain parts of the clause are considered *expressions*, for example, `(print x)` above. An expression can be a single Lisp form or a series of forms implicitly collected with **progn**. An expression is terminated by the next following atom, which is taken to be a keyword. This syntax allows only the first form in an expression to be atomic but makes misspelled keywords more easily detectable. The

loop macro uses print-name equality to compare keywords so that **loop** forms can be written without package prefixes.

Bindings and iteration variable steppings can be performed either sequentially or in parallel, which affects how the stepping of one iteration variable may depend on the value of another. The syntax for distinguishing the two steppings is described with the corresponding clauses. When a set of objects is *in parallel*, all of the bindings produced are performed in parallel by a single lambda binding. Subsequent bindings are performed inside of that binding environment.

Some clauses allow you to specify a *data-type* argument. These arguments are explained in paragraph 15.4, Data Types and Destructuring in the Loop Facility.

Consider the following incorrect example:

```
(let ((x 0))
    (loop while (> x 10)
          (incf x))
  )
```

Assume that this form is supposed to increment `x` each time through the loop and exit when `x` is equal to `10`. Remember that the extended **loop** facility treats multiple Lisp forms as if they were in a **progn** form. In this example, the implied **progn** surrounds both `(> x 10)` *and* `(incf x)`. The value returned by the **progn** (that is, the `incf`) is always true; therefore, the exit condition is never met. This example should have included a **do** clause:

```
(let ((x 0))
    (loop while (> x 10)
          do (incf x))
  )
```

Remember that each clause is introduced by a loop keyword. This use of loop keywords is the syntactic difference between the extended **loop** macro and the Common Lisp **loop**.

Iteration-Driving Clauses

15.3.1 Iteration-driving clauses all create a *variable of iteration*, which is bound locally to the loop and which takes on a new value on each successive iteration. Note that if more than one iteration-driving clause is used in the same loop, several variables are created that all step together through their values; when any of the iterations terminates, the entire loop terminates. Nested iterations are not generated; for those, you need a second **loop** form in the body of the loop. To avoid strange interactions, iteration-driving clauses must precede any clauses that produce *body* code.

Clauses that drive the iteration can be arranged to perform their testing and stepping either in series or in parallel. They are grouped, by default, in series, which allows the stepping computation of one clause to use the just computed values of the iteration variables of previous clauses. They can be made to step *in parallel*, as is the case with the **do** special form, by *joining* the iteration clauses with the keyword **and**. Typically, this form looks something like the following:

```
(loop ... for x = (f) and for y = init then (g x)...)
```

This form sets `x` to `(f)` on every iteration and binds `y` to the value of `init` for the first iteration. On every iteration thereafter, `y` is set to `(g x)`, where `x` still has the value from the previous iteration. Thus, if the calls to `f` and `g` are not order-dependent, this form would be best written as follows:

```
(loop ... for y = init then (g x) for x = (f) ... )
```

This form is used because, as a general rule, parallel stepping has more overhead than sequential stepping. Similarly, note this example:

```
(loop for sublist on some-list
      and for previous =  'undefined then sublist
      ...)
```

The preceding form is equivalent to the following **do** construct:

```
(do ((sublist some-list (cdr sublist))
     (previous 'undefined sublist))
    ((endp sublist) ...)
    ...)
```

However, in terms of stepping, the preceding form would be better written as follows:

```
(loop for previous = 'undefined then sublist
     for sublist on some-list
      ...)
```

When iteration-driving clauses are joined with **and**, if the token following the **and** is not a keyword that introduces an iteration-driving clause, the remaining tokens are assumed to be a syntactical construction of the most recent clause describing parallel bindings; thus, the earlier example showing parallel stepping could have been written as follows:

```
(loop for sublist on some-list
     and previous = 'undefined then sublist
      ...)
```

In iteration-driving clauses, those expressions that are to be evaluated only once are processed in order at the beginning of the form, during the variable-binding phase, whereas those expressions that are to be evaluated each time around the loop are processed in order within the body.

Following are the iteration-driving clauses for the extended **loop** macro. Note that there is no difference between **for** and **as**; select the one that you feel more comfortable with. Optional parts are enclosed in brackets.

for *var* [*data-type*] **in** *expr1* [by *expr2*]
as *var* [*data-type*] **in** *expr1* [by *expr2*]

These clauses iterate over each of the elements in the list *expr1*. If the by subclause is present, *expr2* is evaluated once on entry to the loop to supply the function to be used to obtain successive sublists, instead of using **cdr**. Note also that **loop** uses a **null** rather than an **atom** test to implement the exit condition.

for *var* [*data-type*] **on** *expr1* [**by** *expr2*]
as *var* [*data-type*] **on** *expr1* [**by** *expr2*]

These clauses are like the previous **for** format, except that *var* is set to successive sublists of the list instead of successive elements. Note that because *var* is always a list, it is not meaningful to specify a *data-type* unless *var* is a *destructuring pattern*, as described in paragraph 15.4, Data Types and Destructuring in the Loop Facility. Note also that **loop** uses a **null** rather than an **atom** test to implement the exit condition.

for *var* [*data-type*] = *expr*
as *var* [*data-type*] = *expr*

On each iteration, *expr* is evaluated and *var* is set to the result.

for *var* [*data-type*] = *expr1* **then** *expr2*
as *var* [*data-type*] = *expr1* **then** *expr2*

The *var* argument is bound to *expr1* when the loop is entered and set to *expr2* (reevaluated) at all but the first iteration. Since *expr1* is evaluated during the binding phase, it cannot reference other iteration variables set before it; for that, use the following data type.

for *var* [*data-type*] **first** *expr1* **then** *expr2*
as *var* [*data-type*] **first** *expr1* **then** *expr2*

These clauses set *var* to *expr1* on the first iteration and to *expr2* (reevaluated) on each subsequent iteration. The evaluation of both expressions is performed *inside* of the **loop** binding environment, before the **loop** body. This arrangement allows the first value of *var* to come from the first value of some other iteration variable, allowing such constructs as the following:

```
(loop for term in poly
      for ans first (car term)
              then (gcd ans (car term))
      finally (return ans))
```

for *var* [*data-type*] {**from** | **downfrom** | **upfrom**} *expr1*
[{**to** | **downto** | **below** | **above**} *expr2*] [**by** *expr3*]
as *var* [*data-type*] {**from** | **downfrom** | **upfrom**} *expr1*
[{**to** | **downto** | **below** | **above**} *expr2*] [**by** *expr3*]

These clauses perform numeric iteration. The *var* argument is initialized to *expr1* and on each succeeding iteration is incremented by *expr3* (whose default is 1). If the **to** phrase is given, the iteration terminates when *var* becomes greater than *expr2*. Each of the expressions is evaluated only once, and the **to** and **by** phrases can be written in either order. The **downto** clause can be used instead of **to**, in which case *var* is decremented by the step value, and the end-test is adjusted accordingly. If **below** is used instead of **to**, or **above** instead of **downto**, the iteration is terminated before *expr2* is reached, rather than after. Note that you must use the **to** variant appropriate for the specified direction of stepping; that is, the code does not work if *expr3* is negative or 0. If no limit-specifying clause is given, then the direction of the stepping can be specified as decreasing by using **downfrom** instead of **from**. The **upfrom** clause can also be used instead of **from**; it forces the stepping direction to be increasing. The *data-type* defaults to **fixnum**.

for *var* [*data-type*] **being** *expr* **and its** *path* ...
as *var* [*data-type*] **being** *expr* **and its** *path* ...
for *var* [*data-type*] **being** [**each** | **the**] *path* ...
as *var* [*data-type*] **being** [**each** | **the**] *path* ...

These clauses provide a user-definable iteration facility. The *path* argument names the manner in which the iteration is to be performed. The ellipsis indicates where various path-dependent preposition/expression pairs can appear. See paragraph 15.7, Iteration Paths, for more details.

repeat *expression*

This clause evaluates *expression* during the variable-binding phase and causes the **loop** to iterate that many times. The *expression* argument is expected to evaluate to a fixnum. If *expression* evaluates to 0 or a negative result, the body code is not executed.

Bindings

15.3.2 The **with** keyword can be used to establish initial bindings; that is, variables that are local to the loop but are set only once, rather than on each iteration. The **with** clause has the following format:

with *var1* [*data-type*] [= *expr1*]
{**and** *var2* [*data-type*] [= *expr2*]...}*

If no *expr* is given, the variable is initialized to the appropriate value for its data type, usually **nil**. The **with** bindings linked by **and** are performed in parallel; those not linked are performed sequentially. For example:

```
(loop with a = (foo) and b = (bar) and c
      ...)
```

The preceding form binds variables as follows:

```
(let ((a (foo))
      (b (bar))
       c)
     ...)
```

By contrast, note the following form:

```
(loop with a = (foo) with b = (bar a) with c ...)
```

This form binds variables as follows:

```
(let ((a (foo)))
    (let ((b (bar a)))
        (let (c) ...)))
```

All *exprs* in **with** clauses are evaluated in the order they are written in lambda expressions surrounding the generated **prog**. For example:

```
(loop with a = xa and b = xb
      with c = xc
      for d = xd then (f d)
        and e = xe then (g e d)
      for p in xp
      with q = xq
      ...)
```

The preceding `loop` expression produces the following binding contour. Note that the `for p in xp` clause does not cause `p` to be initially bound to the car of `xp`; that binding is made in the initialization code.

```
(let ((a xa) (b xb))
    (let ((c xc))
        (let ((d xd) (e xe))
            (let ((p nil) (tl xp))
                (let ((q xq))
                  ...)))))
```

Because all expressions in **with** clauses are evaluated during the variable-binding phase, they are best placed near the front of the **loop** form for stylistic reasons.

For binding more than one variable with no particular initialization, you can use the following construct:

with ({*var*}*) [({*data-type*}*)] [**and**]

For example:

```
with (i j k tl t2) (fixnum fixnum fixnum)...
```

A slightly shorter way of writing this is the following:

```
with (i j k) fixnum and (tl t2) ...
```

These are cases of ***destructuring***, which **loop** handles specially; destructuring and data type keywords are discussed in paragraph 15.4, Data Types and Destructuring in the Loop Facility.

Occasionally, a variable must *not* be given a local type declaration for various implementational reasons. If this is necessary, you can use the **nodeclare** clause:

nodeclare ({*var*}*)

The variables in *var* are noted by **loop** as not requiring local type declarations. Consider the following:

```
(proclaim '(special k) '(fixnum k))
(defun foo (lst)
   (loop for x in lst
         as k fixnum = (f x) ...))
```

If `k` did not have the `fixnum` data-type keyword given for it, then `loop` would bind it to **nil**, which could cause some compilers to complain. On the other hand, the `fixnum` keyword also produces a local `fixnum` declaration for `k`; since `k` is special, some compilers will complain (or error out). The solution is to use **nodeclare**:

```
(defun foo (lst)
   (loop nodeclare (k)
      for x in lst
         as k fixnum = (f x) ...))
```

The preceding form tells `loop` not to make that local declaration. The `nodeclare` clause must come *before* any reference to the variables so noted. Positioning it incorrectly causes it to not take effect.

Loop Macro Entrance and Exit Forms

15.3.3 The following are the descriptions for the *prologue* and *epilogue* forms to the loop macro.

initially *expression*
This clause puts *expression* into the *prologue* of the iteration. This expression is evaluated before any other initialization code, other than the initial bindings. For the sake of good style, the **initially** clause should be placed after any **with** clauses but before the main body of the loop.

finally *expression*
This clause puts *expression* into the *epilogue* of the loop, which is evaluated when the iteration terminates (other than by an explicit **return**). For stylistic reasons, then, this clause should appear last in the loop body. Note that certain clauses can generate code that terminates the iteration without running the epilogue code; this behavior is noted with those clauses. (The most notable of these are those described in paragraph 15.3.7, Aggregated Boolean Tests.) These clauses can be used to cause the loop to return values as in the following example:

```
(loop for n in lst
      sum n into the-sum
      count t into the-count
      finally (return (/ the-sum the-count)))
```

Loop Macro Body Clauses

15.3.4 The **do** clauses allow you to include arbitrary code in the body of a loop.

do *expression*
doing *expression*
With the **do** and **doing** forms, *expression* is evaluated each time through the loop.

Accumulation Values

15.3.5 The following clauses accumulate, in some manner, a return value for the iteration. The general form is the following:

type-of-collection expr [*data-type*] [**into** *var*]

In this form, *type-of-collection* is a **loop** keyword, and *expr* is the object being *accumulated*. If no **into** form is specified, then the accumulation is returned when the **loop** terminates. If there is an **into**, then when the epilogue of the **loop** is reached, *var* (a variable automatically bound locally in the loop) is set to the accumulated result and may be used by the epilogue code. In this way, a user can accumulate and somehow pass back multiple values from a single **loop** or use them during the loop. It is safe to refer to these variables during the loop, but they should not be modified until the epilogue code of the loop is reached.

Consider the following example:

```
(loop for x in list
     collect (foo x) into foo-list
     collect (bar x) into bar-list
     collect (baz x) into baz-list
     finally (return (list foo-list bar-list baz-list)))
```

The following code has the same effect as the preceding:

```
(do ((g0001 list (cdr g0001))
     (x)
     (foo-list)
     (bar-list)
     (baz-list))
    ((null g0001)
     (list (nreverse foo-list)
           (nreverse bar-list)
           (nreverse baz-list)))
  (setq x (car g0001))
  (push (foo x) foo-list)
  (push (bar x) bar-list)
  (push (baz x) baz-list))
```

However, in this example **loop** arranges to form the lists in the correct order, obviating the **nreverses** at the end and allowing the lists to be examined during the computation.

collect *expr* [**into** *var*]
collecting *expr* [**into** *var*]

This form causes the values of *expr* on each iteration to be collected into a list.

nconc *expr* [**into** *var*]
nconcing *expr* [**into** *var*]
append *expr* [**into** *var*]
appending *expr* [**into** *var*]

These forms are like **collect**, but the results are put together by **nconc** or **append**, as appropriate. For example:

```
(loop for i from 1 to 3
     nconc (list i (* i i)))
 => (1 1 2 4 3 9)
```

count *bool-expr* [**into** *var*] [*data-type*]
counting *bool-expr* [**into** *var*] [*data-type*]

With **count** and **counting**, if *bool-expr* evaluates non-nil, a counter is incremented. The *data-type* defaults to **fixnum**.

sum *expr* [*data-type*] [**into** *var*]
summing *expr* [*data-type*] [**into** *var*]

This form evaluates *expr* on each iteration and accumulates the sum of all the values. The *data-type* argument defaults to **number**, which, for all practical purposes, is **notype**. Note that specifying *data-type* implies that *both* the sum and the number being summed (the value of *expr*) are of that type.

maximize *expr* [*data-type*] [**into** *var*]
minimize *expr* [*data-type*] [**into** *var*]

These forms compute the maximum (or minimum) of *expr* over all iterations. The *data-type* argument defaults to **number**. Note that if the loop iterates zero times or if conditionalization prevents the code of this clause from being executed, the result is meaningless. If **loop** can determine that the arithmetic being performed is not contagious (by virtue of *data-type* being **fixnum**, **flonum**, or **small-flonum**), then it may choose to code this clause by performing an arithmetic comparison rather than calling either **max** or **min**. As with the **sum** clause, specifying *data-type* implies that both the result of the **max** or **min** operation and the value being maximized or minimized will be of that type.

Not only can there be multiple accumulations in a **loop**, but a single accumulation can come from multiple places *within the same* **loop** *form*. Obviously, the types of the collection must be compatible. The **collect**, **nconc**, and **append** forms can be mixed, as can **sum** and **count**, and **maximize** and **minimize**. For example:

```
(loop for x in '(a b c)
      for y in '((1 2) (3 4) (5 6))
      collect x
      append y)
=> (a 1 2 b 3 4 c 5 6)
```

The following computes the average of the entries in the list `list-of-frobs`:

```
(loop for x in list-of-frobs
      count t into count-var
      sum x into sum-var
      finally (return (/ sum-var count-var)))
```

End-Tests

15.3.6 The following clauses can be used to provide additional control over when the iteration is terminated, possibly running exit code (due to execution of **finally**) and possibly returning a value (for example, from **collect**).

while *expr*

With this clause, if *expr* evaluates to **nil**, the loop is exited, performing exit code (if any) and returning any accumulated value. The test is placed in the body of the loop where it is written. It may appear between sequential **for** clauses.

until *expr*

This clause is identical to **while** (**not** *expr*).

This test may be needed, for example, to step through a strange data structure, as in the following form:

```
(loop until (top-of-concept-tree? concept)
      for concept = expr then (superior-concept concept)
     ...)
```

Note that the placement of the **until** clause before the **for** clause is valid in this case because of the definition of this particular variant of **for**, which *binds* **concept** to its first value rather than setting it from inside the **loop**.

loop-finish Macro

This macro causes the iteration to terminate *normally*, the same as implicit termination by an iteration-driving clause, or by use of **while** or **until**. The epilogue code (if any) is run, and any implicitly collected result is returned as the value of the **loop**. For example:

```
(loop for x in '(1 2 3 4 5 6)
      collect x
      do (when (= x 4) (loop-finish)))
=> (1 2 3 4)
```

This particular example could also be written with `until (= x 4)` in place of the `do` clause.

Aggregated Boolean Tests

15.3.7 Aggregated Boolean tests are clauses that all perform some kind of test and may immediately terminate the iteration, depending on the result of that test.

always *expr*
This test causes the loop to return **t** if *expr* always evaluates non-**null**. If *expr* evaluates to **nil**, the loop immediately returns **nil** without running the epilogue code (if any); otherwise, **t** is returned when the loop finishes after the epilogue code has been run.

never *expr*
This test causes the loop to return **t** if *expr* never evaluates non-**null**. This test is equivalent to **always** (**not** *expr*).

thereis *expr*
If *expr* evaluates non-**nil**, then the iteration is terminated and the non-**nil** value is returned without running the epilogue code.

Conditionalizing Clauses

15.3.8 These clauses can be used to *conditionalize* the consequence clause. They can precede any of the side-effecting or value-producing clauses, such as **do**, **collect**, **always**, or **return**.

when *expr true-consequence-clause* [**else** *false-consequence-clause*]
if *expr true-consequence-clause* [**else** *false-consequence-clause*]
If *expr* evaluates to true, the *true-consequence-clause* is executed; otherwise, it is skipped.

unless *expr consequence-clause*
This test is equivalent to (**when** (**not** *expr*)).

Multiple conditionalization clauses can appear in sequence. If one test fails, then any following tests in the immediate sequence, as well as the clause being conditionalized, are skipped.

Multiple clauses can be conditionalized under the same test by joining them with **and**, as in the following form:

```
(loop for i from a to b
      when (zerop (mod i 3))
        collect i and do (print i))
```

This form returns a list of all multiples of `3` from `a` to `b` (inclusive) and prints them as they are being collected.

If-then-else conditionals can be written using the **else** keyword:

```
(loop for i from a to b
      when (oddp i)
        collect i into odd-numbers
      else collect i into even-numbers)
```

Multiple clauses can appear in an **else** phrase, using **and** to join them.

Conditionals can be nested. For example:

```
(loop for i from a to b
      when (zerop (remainder i 3))
        do (print i)
        and when (zerop (remainder i 2))
              collect i)
```

This form returns a list of all multiples of 6 from `a` to `b` and prints all multiples of `3` from `a` to `b`.

When **else** is used with nested conditionals, the *dangling else* ambiguity is resolved by matching the **else** with the innermost **when** not already matched with an **else**. The following is a complicated example:

```
(loop for x in lst
     when (atom x)
       when (member x *distinguished-symbols* :test #'eq)
            do (process1 x)
       else do (process2 x)
     else when (member (car x) *special-prefixes* :test #'eq)
               collect (process3 (car x) (cdr x))
               and do (memorize x)
             else do (process4 x))
```

Useful with the conditionalization clauses is the **return** clause, which causes an explicit return of its *argument* as the value of the **loop** macro, bypassing any epilogue code. For example:

when *expr1* **return** *expr2*

The preceding form is equivalent to the following:

when *expr1* **do** (**return** *expr2*)

If you conditionalize one of the *aggregated Boolean value* clauses, the test that would cause the iteration to terminate early is not to be performed unless the condition succeeds. For example:

```
(loop for x in lst
     when (significant-p x)
       do (print x)
          (princ "is significant.")
       and thereis (extra-special-significant-p x))
```

This form does not make the `extra-special-significant-p` check unless the `significant-p` check succeeds.

The format of a conditionalized clause is typically something like the following:

when *expr1 keyword expr2*

If *expr2* is the keyword **it**, then a variable is generated to hold the value of *expr1*, and that variable is substituted for *expr2*. For example:

when *expr* **return it**

The preceding form is equivalent to the following clause:

thereis *expr*

Furthermore, you can collect all non-**null** values in an iteration by using the following:

when *expression* **collect it**

If multiple clauses are joined with **and**, the **it** keyword can be used only in the first. If multiple **whens**, **unlesses**, and/or **ifs** occur in sequence, the value substituted for **it** is that of the last test performed. The **it** keyword is not recognized in an **else** phrase.

Miscellaneous Clauses

15.3.9 The following are useful miscellaneous clauses.

named *name*

This clause gives a name of *name* to the **block** that **loop** generates so that you can use the **return-from** form to return explicitly out of that particular **loop**:

```
(loop named sue
      ...
   do (loop ... do (return-from sue value)
        ...) ...)
```

The **return-from** form shown causes *value* to be immediately returned as the value of the outer **loop**. Only one name can be given to any particular **loop** construct.

return *expression*

This clause immediately returns the value of *expression* as the value of the loop without running the epilogue code. This clause is most useful with some sort of conditionalization, as discussed previously. Unlike most of the other clauses, **return** is not considered to *generate body code*, so it is allowed to occur between iteration clauses, as in the following:

```
(loop for entry in list
      when (not (numberp entry))
          return (error ...)
      as frob = (times entry 2)
     ...)
```

If instead you want the loop to have a return value when it finishes normally, place a call to the **return** function in the epilogue with the **finally** clause.

Data Types and Destructuring in the Loop Facility

15.4 In many of the clause descriptions, an optional data type is shown. In this case, a *data type* must be a symbol to be recognized by **loop**. Data types are used for declaration and initialization purposes. For example:

```
(loop for x in l
      maximize x flonum into the-max
      sum x flonum into the-sum
      ...)
```

In this example, the `flonum` data-type keyword for the `maximize` clause indicates that the result of the max operation, and its *argument* `(x)`, are both to be floating-point numbers; hence, `loop` can choose to code this operation specially because it knows there can be no contagious arithmetic. The floating-point data-type keyword for the `sum` clause behaves similarly and causes `the-sum` to be correctly initialized to 0.0 rather than 0. The floating-point keywords also cause the variables `the-max` and `the-sum` to be declared to be floating-point numbers. In general, a numeric data type more specific than **number**, whether explicitly specified or defaulted, is considered by **loop** to be permission to generate code using type-specific arithmetic functions where reasonable. The following data-type keywords are recognized by **loop** (others can be defined; for those, consult the source code).

fixnum — A small integer

flonum — A single-precision floating-point number

small-flonum — A short floating-point number

integer — Any integer (no range restriction)

number — Any number

notype — Unspecified type (that is, anything not specified in the preceding keywords)

Note that explicit specification of a nonnumeric type for a numeric operation (such as the **summing** clause) may cause a variable to be initialized to **nil** when it should be 0. If local data-type declarations must be inhibited, use the **nodeclare** clause.

Destructuring provides you with the ability to *simultaneously* assign or bind multiple variables to components of a list structure. The most common use of destructuring is for association list processing. For example:

```
(loop for (key . datum) in my-alist
   ...)
```

The preceding form sets `key` to the key value and `datum` to the data value for each element in the association list `my-alist`.

You can specify the data types of the components of a pattern by using a corresponding pattern of the data-type keywords in place of a single data-type keyword. This syntax remains unambiguous because wherever a data-type keyword is possible, a **loop** keyword is the only other possibility. For example:

```
(loop for (i j . k) (fixnum fixnum . fixnum) in l ...)

(loop for x in l
     for i fixnum = (car x)
     and j fixnum = (cadr x)
     and k fixnum = (cddr x)
     ...)
```

To allow some abbreviation of the data-type pattern, an atomic component of the data-type pattern is considered to state that all components of the corresponding part of the variable pattern are of that type. That is, the previous form could be written as follows:

```
(loop for (i j . k) fixnum in l ...)
```

This generality allows binding of multiple typed variables in a reasonably concise manner, as in the following:

```
(loop with (a b c) and (i j k) fixnum ...)
```

This form binds `a`, `b`, and `c` to `nil` and `i`, `j`, and `k` to 0 for use as temporaries during the iteration; it also declares `i`, `j`, and `k` to be fixnums for the benefit of the compiler. Consider another example:

```
(defun map-over-properties (fn symbol)
  (loop for (propname propval) on (plist symbol) by 'cddr
        do (funcall fn symbol propname propval)))
```

The preceding form maps `fn` over the properties on `symbol`, giving it arguments of the symbol, the property name, and the value of that property.

Loop Synonyms

15.5 The following macro can be used to give synonyms to **loop** keywords, therefore allowing you to change keyword names to better fit your style.

define-loop-macro *keyword* Macro

This macro can be used to make *keyword*, a **loop** keyword (such as **for**), into a Lisp macro that can introduce a **loop** form. For example:

```
(define-loop-macro for)
```

After evaluating the preceding form, you can now write an iteration as follows:

```
(for i from 1 below n do ...)
```

This facility exists primarily for diehard users of a predecessor of **loop**. Its unconstrained use is not recommended because it tends to decrease the transportability of the code and needlessly uses up a function name.

However, note also the following example:

```
(loop for x being '(a b c . d) and its cdrs collect x)
=> ((a b c . d) (b c . d) (c . d) d)
```

The clauses **his**, **her**, or **their** can be substituted for the **its** keyword, as can **each**.

Very often, iteration paths step internal variables that you do not specify, such as an index into a data structure. Although in most cases you do not wish to be concerned with such low-level matters, it is occasionally useful to have a handle on such things. The **loop** macro provides the **using** *prepositional phrase* so that you can provide a variable name to be used as an *internal* variable by an iteration path. The **using** phrase is placed with the other phrases associated with the path and contains any number of keyword/variable-name pairs:

```
(loop for x being the array-elements of a using (index i)
      ...)
```

The preceding form indicates that the variable `i` should be used to hold the index of the array being stepped through. The particular keywords that can be used are defined by the iteration path; the `index` keyword is recognized by all **loop** sequence paths. Note that any individual `using` phrase applies to only one path; it is parsed along with the prepositional phrases. An error is produced if the path does not call for a variable using that keyword.

By special dispensation, if a *path* is not recognized, then the **default-loop-path** path is invoked upon a syntactic transformation of the original input. For example:

for *var* **being** *frob*

The **loop** fragment in the preceding form is taken as if it were the following:

for *var* **being default-loop-path in** *frob*

Similarly, consider the following:

for *var* **being** *expr* **and its** *frob* ...

The preceding form is taken as if it were the following:

for *var* **being** *expr* **and its default-loop-path in** *frob*

Thus, this *undefined path hook* works only if the **default-loop-path** path is defined. Obviously, the use of this hook is competitive, because only one such hook can be in use, and the potential for syntactic ambiguity exists if *frob* is the name of a defined iteration path. This feature is not for casual use; it is intended for large systems that need to use special syntaxes for features they provide.

Predefined Paths

15.7.1 The **loop** macro comes with two predefined iteration path functions: one implements an iteration path facility like **mapatoms**, and the other is used for defining iteration paths for stepping through sequences.

The Interned-Symbols Path

15.7.1.1 The **interned-symbols** iteration path is like a **mapatoms** for **loop**.

for *var* **being** [**each** | **the**] **interned-symbols** [**in** *package*]
This form iterates over all of the symbols in the current package and its superiors. This is the same set of symbols over which **mapatoms** iterates, although not necessarily in the same order. The *package* argument specifies the particular package to look in, which is like giving a second argument to **mapatoms**.

for *var* **being** [**each** | **the**] **local-interned-symbols** [**in** *package*]
This clause is like the preceding **for** clause with **interned-symbols** except that the clause with **local-interned-symbols** restricts the iteration so that it affects only the package specified, but not its superiors:

```
(loop for sym being the local-interned-symbols in package
     ...)
```

Consider the following example:

```
(defun my-apropos (sub-string &optional (pkg package))
   (loop for x being the interned-symbols in pkg
      when (search sub-string x)
        when (or (boundp x) (fboundp x) (symbol-plist x))
          do (print-interesting-info x)))
```

A package specified with the **in** preposition can be anything acceptable to the **find-package** function. The code generated by this path contains calls to internal **loop** functions, with the effect that it is transparent to changes in the implementation of packages.

Sequence Iteration Path

15.7.1.2 One very common form of iteration is performed over the elements of an object that is accessible by means of an integer index. The **loop** macro defines an iteration path function for doing this in a general way and provides a simple interface to allow you to define iteration paths for various kinds of *indexable* data.

define-loop-sequence-path *pathnames fetch-fn size-fn* &optional *sequence-type default-var-type* — Macro

This macro defines an iteration path. The argument *pathnames* is either an atomic pathname or a list of pathnames. (*Pathnames* here is the name of an iteration path; it has nothing to do with accessing files, which is described in the *Explorer Input/Output Reference* manual.) The *fetch-fn* argument is a function of two arguments: the sequence and the index of the item to be obtained. (Indexing is assumed to be zero-origined.) The *size-fn* argument is a function of one argument—the sequence; it should return the number of elements in the sequence. The *sequence-type* argument is the name of the data type of the sequence, and *default-var-type* is the name of the data type of the elements of the sequence.

The array-manipulation primitives are used to define both **array-element** and **array-elements** as iteration paths:

```
(define-loop-sequence-path (array-element array-elements)
   aref length)
```

Then, the following **loop** clause steps *var* over the elements of *array*, starting from 0:

for *var* **being the array-elements of** *array*

The sequence path function also accepts **in** as a synonym for **of**.

The range and stepping of the iteration can be specified with the use of all of the same keywords that are accepted by the **loop** arithmetic stepper:

(`for` *var* `from` . . .)

The keywords are **by**, **to**, **downto**, **from**, **downfrom**, **below**, and **above**, and they are interpreted in the same way. For example:

```
(loop for var being the array-elements of array
            from 1 by 2
     ...)
```

The preceding form steps *var* over all of the odd elements of `array`. Now consider the following:

```
(loop for var being the array-elements of array
      downto 0
     ...)
```

The preceding form steps in *reverse* order.

All such sequence iteration paths allow you to specify the variable to be used as the index variable with the **using** prepositional phrase and the **index** keyword.

Defining Paths

15.7.2 In addition to the code that defines the iteration, a **loop** iteration clause (for example, a **for** or **as** clause) produces variables to be bound and preiteration (*prologue*) code. This breakdown allows the creation of a user interface to **loop** that need not depend on or know about the internals of **loop**. To complete this separation, the iteration path mechanism parses the clause before giving it to the user function that is to return those items. A function to generate code for a path can be declared to **loop** with the **define-loop-path** function.

define-loop-path *pathname-or-names path-function list-of-allowable-prepositions* &rest *data* Macro

This macro defines *path-function* to be the handler for the path(s) *pathname-or-names*, which can be either a symbol or a list of symbols. Such a handler should follow the conventions described below. The *data* arguments are optional; they are passed to *path-function* as a list. The handler is called with the following arguments:

path-name — The name of the path that caused the path function to be invoked.

variable — The iteration variable.

data-type — The data type supplied with the iteration variable, or **nil** if none was supplied.

prepositional-phrases — This is a list with entries of the form (*preposition expression*), in the order in which they were collected. This can also include entries supplied implicitly (for example, an **of** phrase when the iteration is inclusive and an **in** phrase for the **default-loop-path** path); the ordering shows the order of evaluation that should be followed for the expressions.

inclusive? — This argument is **t** if *variable* should have the starting point of the path as its value on the first iteration (by virtue of being specified with syntax like **for** *var* **being** *expr* **and its** *pathname*), or **nil** otherwise. When **t**, *expr* appears in *prepositional-phrases* with the **of** preposition. For example, the following clause receives the *prepositional-phrases* of (**of foo**):

```
for x being foo and its cdrs
```

allowed-prepositions — This argument is the list of allowable prepositions declared for the pathname that caused the path function to be invoked. It and *data* (immediately below) can be used by the path function such that a single function can handle similar paths.

data — This argument is the list of data declared for the pathname that caused the path function to be invoked. The *data* argument can, for instance, contain a canonicalized pathname, or a set of functions or flags to aid the path function in determining what to do. In this way, the same path function may be able to handle different paths.

The handler should return a list of either six or ten elements:

variable-bindings — This is a list of variables that need to be bound. The entries in it may be of the form *variable*, (*variable expression*), or (*variable expression data-type*). Note that it is the responsibility of the handler to make sure the iteration variable is bound. All of these variables are bound in parallel; if initialization of one depends on others, it should be performed with a **setf** in the *prologue-forms*. Returning only the variable without any initialization expression is not allowed if the variable is a destructuring pattern.

prologue-forms — This is a list of forms that should be included in the **loop** prologue.

the four items of the iteration specification — These are the four items described previously, *pre-step-endtest*, *steps*, *post-step-endtest*, and *pseudo-steps*.

another four items of iteration specification — If these four items are given, they apply to the first iteration, and the previous four apply to all succeeding iterations; otherwise, the previous four apply to *all* iterations.

The following are the routines that are used by **loop** to compare keywords for equality. In all cases, a *token* can be any Lisp object, but a *keyword* is expected to be an atomic symbol.

sys:loop-tequal *token keyword* Function

This is the **loop** token comparison function. The *token* argument is any Lisp object; *keyword* is the keyword it is to be compared with. This macro returns true if they represent the same token, using **string-equal** for comparing.

sys:loop-tmember *token keyword-list* Function

This function is the **member** variant of **sys:loop-tequal.**

sys:loop-tassoc *token keyword-alist* Function

This function is the **assoc** variant of **sys:loop-tequal.**

If you want an iteration path function to make an internal variable accessible to the user, have the function call the following function instead of **gensym.**

sys:loop-named-variable *keyword* Function

This function should be called only from within an iteration path function. If *keyword* has been specified in a **using** phrase for this path, the corresponding variable is returned; otherwise, **gensym** is called and that new symbol is returned. Within a given path function, this routine should be called only once for any given keyword.

If you specify a **using** preposition containing any keywords for which the path function does not call **sys:loop-named-variable, loop** informs you of the error.

An Example Path Definition

15.7.3 The following is an example function that defines the **string-characters** iteration path. This path steps a variable through all the characters of a string. It accepts the following format:

```
(loop for var being the string-characters of str ...)
```

The function is defined to handle the path by using the following:

```
(define-loop-path string-characters string-chars-path (of))
```

The following is the example function:

```
(defun string-chars-path (path-name variable data-type
                                    prep-phrases inclusive?
                                    allowed-prepositions data
                                    &aux (bindings nil)
                                         (prologue nil)
                                         (string-var (gensym))
                                         (index-var (gensym))
                                         (size-var (gensym)))
 (declare (ignore allowed-prepositions data))
 ;; To iterate over the characters of a string, we need to save the string, save the size
 ;; of the string, step an index variable through that range, setting the user's variable to
 ;; the character at that index. Default the data-type of the user's variable:
 (when ((null data-type) (setf data-type 'fixnum)))
 ;; Since we support exactly one "preposition," which is required, the following check
 ;; suffices:
 (when (null prep-phrases)
        (error "OF missing in ~S iteration path of ~S"
                        path-name variable))
 ;; We do not support "inclusive" iteration:
 (when (not (null inclusive?))
    (error
      "Inclusive stepping not supported in ~S path ~
       of ~S (prep phrases = ~:S)"
      path-name variable prep-phrases))
 ;; Set up the bindings.
 (setf bindings (list (list variable nil data-type)
                       (list string-var (cadar prep-phrases))
                       (list index-var 0 'fixnum)
                       (list size-var 0 'fixnum)))
 ;; Now set the size variable.
 (setf prologue (list `(setf ,size-var (length ,string-var))))
 ;; And return the appropriate values, explained below.
 (list bindings
       prologue
       `(= ,index-var ,size-var)
       nil
       nil
       (list variable `(aref ,string-var ,index-var)
             index-var `(1+ ,index-var))))
```

The first element of the returned list is the bindings. The second is a list of forms to be placed in the *prologue*. The remaining elements specify how the iteration is to be performed.

This example is a particularly simple case for two reasons: the actual variable of iteration, `index-var`, is purely internal (created by `gensym`), and the stepping of it (1+) is such that it can be performed safely without an end-test. Thus, `index-var` can be stepped immediately after the setting of the user's variable, causing the iteration specification for the first iteration to be identical to the iteration specification for all remaining iterations. This procedure is advantageous from the standpoint of the optimizations that **loop** is able to perform, although it is frequently not possible due to the semantics of the iteration (for example, **for** *var* **first** *expr1* **then** *expr2*) or to subtleties of the stepping. It is safe for this path to step the user's variable in the *pseudo-steps* (the fourth item of an iteration specification) rather than the *real* steps (the second), because the step value can have no dependencies on any other (user) iteration variables. Using the pseudo-steps generally increases efficiency.

If you want the index variable in the above definition to be user-accessible through the **using** phrase feature with the **index** keyword, the function must be changed in two ways. First, `index-var` should be bound to `(sys:loop-named-variable 'index)` instead of `(gensym)`. Second, the efficiency hack of stepping the index variable ahead of the iteration variable must not be done. This is effected by changing the last form to be the following:

```
(list bindings prologue
     nil
     (list index-var `(1+ ,index-var))
     `(= ,index-var ,size-var)
     (list variable `(aref ,string-var ,index-var))
     nil
     nil
     `(= ,index-var ,size-var)
     (list variable `(aref ,string-var ,index-var)))
```

Note that although the second `` `(= ,index-var ,size-var) `` could have been placed earlier (where the second `nil` is), it is best for it to match up with the equivalent test in the first iteration specification grouping.

FUNCTIONS

16

Function Terms and Concepts

16.1 A *function* is a Lisp object that can be applied to arguments. Because functions are Lisp objects, you can manipulate them in the usual ways: you can pass them as arguments, return them as values, and make other Lisp objects refer to them. However, to understand the use of functions, you must first know how to define them (by using lambda expressions) and how to reference them (by using function specs).

Lambda Expressions

16.1.1 A *lambda expression* defines a function without naming it. The lambda expression is a list whose first element is **lambda** and whose remaining elements describe a function. It has two main purposes: to define how to match up *arguments* (passed by the caller) to the *parameters* in the function definition, and to define the control structure of the function itself. A Common Lisp lambda-list has the following syntax:

(**lambda** *lambda-list* {*declaration*|*doc-string*}* {*form*}*)

where a Common Lisp *lambda-list* is the following:

({*var*}*
    [&optional {*var*|(*var* [*initform* [*supplied-p*]])}*]
    [&rest *var*]
    [&key {*var*| ({*var*| (*keyword var*)} [*initform* [*supplied-p*]])}*
    [&allow-other-keys]]
    [&aux {*var*| (*var* [*initform*])}*])

The *lambda-list* contains the names of the parameters to be used by the lambda expression. Each variable name must be unique; an error is signaled if the *var* names and the *supplied-p* names are not all different. (On the Explorer system, you can have more than one parameter named `ignore` or `ignored`.) These parameters are bound during execution of the body *forms* and are then released. The result of the last body *form* is returned as the value of the invocation of the lambda expression. Thus, a call to a lambda expression that takes two arguments and returns them in a list is as follows:

```
((lambda (a b)
        (list a b))
 2 3) => (2 3)
```

Note that the lambda expression itself, including the lambda list and the body form, are enclosed in parentheses and constitute the first element of the function call form, rather than a function name. When typed in the Lisp Listener, this lambda expression is translated to a function object by the function **function** and applied to the remaining arguments.

Lambda-List Keywords

16.1.2 *Lambda-list keywords* are used in lambda expressions to indicate that the following parameter (or parameters) is to be treated in a certain way. In a lambda expression, these keywords and their associated parameters always come after the required parameters (with the exception of &quote and &special, which are not Common Lisp).

lambda-list-keywords *[c]* Constant

This constant contains a list of available lambda-list keywords (which includes those used by **defmacro**) in the current Lisp implementation.

lambda-parameters-limit *[c]* Constant

This constant contains a positive integer indicating how many different parameters are allowed in one lambda list. This integer is an upper exclusive bound; therefore, since **lambda-parameters-limit** has a value of 64 on the Explorer system, you can have up to 63 different parameters in one lambda list.

The following are the lambda-list keywords available on the Explorer system:

&optional *[c]* Lambda-List Keyword

If provided, this keyword is placed before &key and &rest keywords. As the name implies, this parameter is optional. If no argument value is specified for an optional parameter, it is bound to **nil**. For example:

```
((lambda (a &optional b)
    (list a b))
  2 3) => (2 3)

((lambda (a &optional b)
    (list a b))
  2) => (2 nil)
```

You can also specify default values for optional parameters by using an *init-form* in a list. The first element of such a list is the parameter, and the second element is the *init-form*. If an argument value is provided for this parameter in the lambda call form, then the *init-form* is not used as the argument value. If no argument is provided, the *init-form* is evaluated and the parameter is bound to the returned value. An *init-form* can refer to parameters to the left of it in the lambda list. For example:

```
((lambda (a &optional (b 4))
    (list a b))
  2) => (2 4)

((lambda (a &optional (b a))
    (list a b))
  1) => (1 1)
```

Within an *init-form* list, you can include a third element (a *supplied-p* parameter) to indicate whether the first element has been provided with an argument value. If the value is provided, the *supplied-p* variable is bound to true. If the first element is not provided with an argument value, the first element is bound to the result of evaluating the *init-form* element and the *supplied-p* variable is bound to **nil**. For example:

```
((lambda (&optional a (b 4 supplied-p) d)
    (list a b supplied-p d))
  1 2)
=> (1 2 t nil)

((lambda (&optional a (b 4 supplied-p) d)
    (list a b supplied-p d))
  1)
=> (1 4 nil nil)
```

In the first example, `b` is provided with an argument value, so it is bound to this argument value, `2`, and the `supplied-p` variable is bound to `t`. In the second example, `b` is not provided with an argument value, so it is bound to its initialization value, `4`, and `c` is bound to `nil`.

&rest *[c]* Lambda-List Keyword

This keyword follows any optional parameters and takes only one parameter, which is bound to a list of all argument values remaining after all required and optional parameters have been bound. If no argument values are provided for this &rest parameter, it is bound to **nil**. For example:

```
((lambda (a &optional b c &rest supplied-c-P)
    (list a b c supplied-c-P))
 1 2 3 4 5 6 7) => (1 2 3 (4 5 6 7))

((lambda (&optional (a 2 supplied-a-P) (c 5 d) &rest e)
    (list a supplied-a-P c d e))
  1) => (1 t 5 nil nil)
```

In the first example, the &rest parameter is bound to the list `(4 5 6 7)`. In the second example, the &rest parameter is not provided with an argument and is thus bound to `nil`.

&key *[c]* Lambda-List Keyword

This keyword specifies that all parameters following it are to be keyword parameters to the body of the lambda expression. Keyword parameters are always optional, whether or not the &optional keyword has previously appeared in the lambda list. Keyword parameters allow *init-forms* and *supplied-p* parameters in the same way as &optional parameters. If a keyword parameter is not supplied an argument value or an *init-form*, then its value defaults to **nil** just like any other optional parameter.

To provide a keyword value in an actual argument list, supply the parameter name, prefixed by a colon, followed by the argument value. Within the body of the lambda expression, the keyword value is accessed by simply using the keyword parameter name (not prefixed by a colon). For example:

```
((lambda (&key a b (c 3 c-specified-p))
     (list a b c c-specified-p))
  :a 1)
=> (1 nil 3 nil)

((lambda (&key a b (c 3 c-specified-p))
    (list a b c c-specified-p))
  :c 1 :a 2 :b 3))
=> (2 3 1 t)
```

Note that in a function call, both required and optional arguments are position dependent. The order for specifying keyword arguments is position independent; however, each position-dependent parameter must receive an argument before any keyword parameters are matched to any arguments. For example:

```
((lambda (a &optional b &key c)
    (list a b c))
  1 :c 2)
=> ERROR
```

In the above example, `a` is bound to `1` and `b` is bound to `:c`. At this point, the remaining argument, `2`, does not specify a legal keyword and corresponding value.

Sometimes you do not want the keyword name to be the same as the variable to which the keyword value is set. For instance, you may want the keyword name to be long and explanatory, but in the body of your source code you prefer to use a shorter name. In this case, you should supply a list containing the keyword name and the variable it is to be known by in place of just the keyword name. For example:

```
((lambda (&key x ((:the-y-value y))((:the-z-value z) 'default-z-value))
   (list x y z))
 :the-y-value 1 :the-z-value 2) => (nil 1 2)

((lambda (&key x ((:the-y-value y))((:the-z-value z) 'default-z-value))
    (list x y z))
    )                                  ; Default all arguments.
=> (nil nil default-z-value)
```

You can also change the descriptive names for all arguments, including keywords, by using `(declare (arglist...))`. See Section 13, Declarations, for details.

&allow-other-keys *[c]* Lambda-List Keyword

This keyword allows the caller to supply keyword arguments even though the lambda expression does not bind them. Typically, these keywords are passed on in a call to another function. Ordinarily, if a keyword and corresponding value are specified in the arguments of a function call but not in the lambda list of the function definition, an error results:

```
((lambda (x y &key z)
    (list x y z))
  1 2 :z 3 :a 4)
=> ERROR
```

You can avoid this error by including an &allow-other-keys lambda-list keyword in the function's lambda list or by including the **:allow-other-keys** keyword with a true value with the arguments in the function call. For example:

```
((lambda (x y &key z &allow-other-keys)
    (list x y z))
  1 2 :z 3 :a 4) => (1 2 3)

((lambda ( x y &key z)
    (list x y z))
  1 2 :allow-other-keys t :a 4) => (1 2 nil)
```

Note that in the last example the value for the `:a` keyword is not used in the function. In fact, **:allow-other-keys** keywords are accessible in a function only if the &allow-other-keys lambda-list keyword is preceded by a &rest lambda-list keyword. For example:

```
((lambda (x &rest y &key z &allow-other-keys)
    (list x y z))
  1 :z 3 :a 4)
=> (1 (:z 3 :a 4) 3)

((lambda (x &rest y &key z)
    (list x y z))
  1 :z 3 :allow-other-keys t :a 4)
=> (1 (:z 3 :allow-other-keys t :a 4) 3)
```

Note that in the above example the keyword `:a` is not a parameter in the lambda expression, but it is not an error because the `&allow-other-keys` keyword is used. Notice that the `&rest` variable, `y` in this case, includes the keyword `:a` and its value. Thus, if you wanted to access the other keywords, you could examine the `&rest` argument value.

&aux *[c]* Lambda-List Keyword

This keyword specifies that the function is to have *auxiliary* variables. In other words, the variables following this keyword are not parameters. They are not bound to any argument value during lambda-list parameter binding. Their purpose is actually to establish local variables within the function as **let*** does. The choice of using &aux rather than **let*** is merely a matter of style. Auxiliary variables can be initialized in the same way as optional parameters, by containing them in a list with an initialization form. For example:

```
((lambda (a b &aux (x 5) (y 3))
    (list a b x y)
  '(0 1) 2) => ((0 1) 2 5 3)
;The following is equivalent to the preceding form:
((lambda (a b)
    (let ((x 5)
          (y 3))
      (list a b x y)))
  '(0 1) 2) => ((0 1) 2 5 3)
```

The following lambda-list keywords are not part of the Common Lisp standard.

&extension

Lambda-List Keyword

On the Explorer system, this keyword means that all preceding parameters operate in accordance with Common Lisp specifications and that all subsequent parameters supply functionality that may be available only on the Explorer system. For the sake of compatibility, &extension cannot appear prior to any required argument. Consider the following parameter list:

```
(make-hash-table (&key test size rehash-size rehash-threshold
       &extension compare-function hash-function actual-size
        number-of-values rehash-function)
 ...)
```

Common Lisp supports the functionality provided by the first four keywords, whereas those parameters that appear after &extension are Explorer extensions. The &extension specifier can also appear before (non-keyword) optional parameters. In the current implementation, &extension serves only as a visual delimiter and has no effect.

&special

Lambda-List Keyword

On the Explorer system, this keyword declares any following parameters and/or auxiliary variables to be special within the scope of the function that uses it. This operation does not apply to keyword parameters. Each variable that follows &special is equivalent to (**declare** (**special** *var*)).

&local

Lambda-List Keyword

On the Explorer system, this keyword turns off a preceding &special declaration for the variables that follow. Each parameter name that follows &local is equivalent to (**declare** (**unspecial** *var*)).

&functional

Lambda-List Keyword

On the Explorer system, this keyword tells the compiler that the value to the next parameter should be a function. If, while being compiled, the caller of this function passes a quoted lambda expression, the compiler knows that the argument is intended to be used as a function definition, rather than as a list. Thus, the argument is compiled. For this keyword to be effective, the compiler must see this function definition before compiling the calling function.

&quote

Lambda-List Keyword

On the Explorer system, this keyword declares that the arguments of the calling form that correspond to the parameters following this keyword are not to be evaluated in the calling form. This is how a special form is created. For this keyword to be effective, the compiler must see the function definition containing the &quote before it compiles the calling function. See paragraph 16.10, Special Forms.

&eval

Lambda-List Keyword

On the Explorer system, this keyword causes the parameters following it to be evaluated despite a preceding &quote lambda-list keyword. See paragraph 16.10, Special Forms.

Function Specs

16.1.3 How do you refer to a function once it has been defined? Typically, we tend to think of a function definition and an associated symbol as one and the same. However, the name of a function need not be a symbol. Various kinds of Lisp objects describe other locations where a function definition can be found. A Lisp object that describes a place to find a function is called a *function spec* (or specification).

The following is a description of all available function specs. Only the first one is Common Lisp standard; all others are Explorer extensions.

symbol *[c]* Function Spec

The function definition is located in the function cell of a symbol. This is the only function spec defined in Common Lisp.

(**:property** *symbol property*) Function Spec

With this function spec, the function definition is located on the property list of the symbol. Explicit application is required to apply this function definition to arguments; for example, both of the following are legal calling forms:

(**funcall** (**get** *symbol property*) *arg*)

(**apply** (**get** *symbol property*) *arg-list*)

Storing functions on property lists is a frequently used technique for dispatching (that is, deciding at run time which function to call, on the basis of input data).

(**:method** *flavor-name operation*) Function Spec
(**:method** *flavor-name method-type operation*) Function Spec
(**:method** *flavor-name method-type operation suboperation*) Function Spec

With these function specs, the function definition is located inside internal data structures of the flavor system and is called automatically as part of handling *operation* on instances of *flavor-name*. See Section 19, Flavors, for details.

(**:handler** *flavor-name operation*) Function Spec

This function spec is a name for the function actually called when an *operation* message is sent to an instance of the flavor specified by *flavor-name*. The difference between **:handler** and **:method** is that the handler can be a method inherited from another flavor or a *combined method* automatically written by the flavor system. Methods are what you define in source files; handlers are not. Note that redefining or encapsulating a handler affects only the named flavor, not any other flavors built out of it. Thus, **:handler** function specs are often used with **trace**, **breakon**, and **advise**. (See the *Explorer Tools and Utilities* manual.)

(**:within** *within-function function-to-affect*) Function Spec

This function spec supplies a hook for gaining access to the environment in which a function is executing. The **:within** specification identifies the function spec that is executed when *function-to-affect* is called within the lexical scope of *within-function*. In this regard, this specification can be used to provide a kind of shadowing of function specs. This function spec works only when *within-function* is interpreted.

For instance, if you want to know what is being **evaled** within a specific function, you could call **breakon** on **eval**, but because of the extensive use of **eval** throughout the software, your system would become unusable. The following **:within** form produces the desired results:

```
(breakon '(:within myfunction eval))
```

Consider the following example:

```
(defun test (x)
    (1+ x))

(test 1) => 2

;;Change 1+ to 1- while inside the test function.
(fdefine '(:within test 1+) '1-)

(test 1) => 0
```

Note that the following form reverts `test` to its original operation:

```
(fdefine '(:within test 1+) '1+)

(test 1) => 2
```

(**:internal** *function-spec number-or-name*) Function Spec

This function spec names an internal function. Some Lisp functions contain internal functions, created either by **#'(lambda...)** or **(function (lambda...))** forms. These internal functions need names when compiled, but they do not have symbols as names; instead they are named by **:internal** function specs. The *function-spec* argument is the name of the containing function. The *number-or-name* argument is a sequence number or an internal name (the name from **flet**, for example); the first internal function that the compiler comes across in a given function is numbered 0, the next 1, and so on. Internal functions are located inside the compiled function object of their containing function.

If a Lisp function uses **flet** to name an internal function, you can use the local name defined with **flet** in the **:internal** function spec instead of a number. The following is an example of such a function:

```
(defun foo (a)
   (flet ((square (x) (* x x)))
      (+ a (square a))))
```

After compiling `foo`, you could use the function spec `(:internal foo square)` to refer to the internal function locally named `square`. You could also use `(:internal foo 0)`. If you have multiple **flets** defining local functions with the same name, only the first can be referred to by name in this way.

(**:location** *pointer*) Function Spec

The function definition of this function spec is stored in the cdr of *pointer*, which can be a locative or a list. This function spec is for pointing at an arbitrary place that cannot be described in any other way. This form of function spec is not useful in **defun** (and related macros) because the Lisp Reader has no printed representation for locative pointers and always creates new lists; these function specs are intended for programs that manipulate functions.

The following is an example of defining a function with a function spec that is not a symbol:

```
(defun (:property foo bar-maker) (thing &optional kind)
      (set-the 'bar thing (make-bar 'foo thing kind)))
```

This `defun` puts a function on the `bar-maker` property of `foo`. Note that to invoke this function, explicit application must be used:

```
(funcall (get 'foo 'bar-maker) 'baz)
```

Actually, symbols and **:property** function specs are the only function specs that can be meaningfully used in a **defun**. The others are created indirectly by the system. Unlike the other kinds of function specs, a symbol by itself *can* be implicitly or explicitly applied to arguments.

If you implicitly apply a symbol to arguments (by making the symbol the first element in an **evaled** list) or explicitly **apply** or **funcall** a symbol to arguments, the symbol's function definition cell is used. For example:

```
(funcall 'print something-to-print)
```

But this is an exception; in general, you cannot apply function specs to arguments. To apply the function definition of a nonsymbolic function spec to arguments, you must use an accessing operation like **function**, which knows how to find the particular function definition of the function spec. In the previous example, the function spec `(:property foo bar-maker)` allows a function definition to be located on the property list of `foo`. Therefore, to apply this function definition to arguments, the accessing operation `get` is used; the following form could also be used:

```
(funcall #'(:property foo bar-maker) 'baz)
```

If the form `(funcall '(:property foo bar-maker) 'baz)` is invoked, it produces an error because `funcall` does not know where to find the appropriate function definition.

Kinds of Functions

16.2 The following are the five kinds of functions that can be defined on the Explorer system:

- *Interpreted* functions are represented as list structures and are interpreted by the Lisp evaluator. For example, an interpreted function is created when a **defun** form is evaluated.
- *Compiled* functions are directly executed by the microcode. A compiled function is created when a **defun** form is compiled. The **compile** function can be used to convert an interpreted function into a compiled function.
- *Microcoded* functions are written in microcode using the microcode assembler and are directly executed by the hardware.
- Various types of Lisp objects can be applied to arguments, but when applied they call another function instead. These types of objects include closures and flavor instances.

- Various types of Lisp objects, when used as functions, do something special in relation to the specific data type. A stack group is an example of such a data object.

Interpreted Functions

16.2.1 An interpreted function is a piece of list structure that represents a program according to the rules of the Lisp interpreter. Unlike other kinds of functions, interpreted functions can be printed out and read back in (they have a printed representation that the Lisp Reader understands); they can be printed in a form that people can understand; and they can be examined with the usual functions for list-structure manipulation.

There are four kinds of interpreted functions: **lambdas**, **named-lambdas**, **substs**, and **named-substs**.

lambda *lambda-list {declaration|doc-string}* {body-form}** *[c]* Interpreted Function

The simplest kind of functions are those defined by **lambda**, which is discussed in paragraph 16.1.1, Lambda Expressions.

The remaining three kinds of functions are all Explorer extensions. It is unlikely that you will want to write them directly. They are created by forms such as **defun** and **defsubst**.

named-lambda *name lambda-list {declaration|doc-string}* {body-form}** Interpreted Function

This interpreted function is like a **lambda** but contains extra information in which the system remembers the function's name, documentation, and other information. Having the function's name in a **named-lambda** function allows the error handler and other tools to give the user more information. Normally, you do not write a **named-lambda** as you do a **lambda**. The **defun** and **flet** forms are somewhat easier to use and, in fact, eventually use **named-lambda**.

If the *name* component is a symbol, then it is the function's name. Otherwise, the *name* component is a list whose first element is the function's name and whose second element is the function's debugging information property list. The name need not be a symbol; it can be any function spec.

subst *lambda-list {declaration|doc-string}* {body-form}** Interpreted Function

This interpreted function is exactly like a **lambda** as far as the interpreter is concerned.

The difference between a **subst** and a **lambda** is the way they are handled by the compiler. A call to a normal function is compiled as a *closed subroutine*; the compiler generates code to compute the values of the arguments and then applies the function to those values. A call to a **subst** is compiled as an *open subroutine*; the compiler incorporates the body forms of the **subst** into the function being compiled, substituting the argument forms for references to the variables in the *lambda-list* of **subst**.

This is a simple-minded but useful facility for *open-coded* or *inline* functions. It is simple-minded because the argument forms can be evaluated numerous times or out of order, so the semantics of a **subst** may not be the same in the interpreter and in the compiler. (A facility for inline expansion that carefully preserves the same semantics as an out-of-line call is provided through the use of the **inline** declaration.) Note that **inline** declarations are supported by Common Lisp, whereas **subst** is not.

named-subst *lambda-list {declaration|doc-string}** *{body-form}** Interpreted Function

This interpreted function is the same as a **subst**, except that it has a name just as a **named-lambda** does. The symbol *name* is interpreted the same way as in a **named-lambda**.

Compiled Functions

16.2.2 The two kinds of compiled functions are *macrocoded* functions and *microcoded* functions. The Lisp compiler converts **lambda** and **named-lambda** functions into macrocoded functions. A macrocoded function has a data type of **compiled-function**. The printed representation for the function **append** looks like the following:

```
#<dtp-function append 1424771>
```

This type of Lisp object is sometimes called a *function entry frame*, or FEF for short. As with car and cdr, the name is historical in origin and does not really mean anything. The object contains Explorer machine code that performs the computation expressed by the function; it also contains a description of the arguments accepted, any constants required, the name, documentation, and other things. Macrocoded functions are full-fledged objects and can be passed as arguments, stored in data structures, and, of course, applied to arguments.

A microcoded function has a data type of **microcode-function**. The printed representation for the function **list** looks like the following:

```
#<dtp-u-entry list 5>
```

Most microcoded operations are accessed as machine instructions. They have a function defined in Lisp that interfaces to that microcoded operation. Microcoded function objects of type **dtp-u-entry** exist only for those operations that take a varying number of arguments, such as **aref** and **list**.

Other Kinds of Functions

16.2.3 A *closure* is a kind of function that contains another function and a set of variable bindings. When the closure is applied, it puts the bindings into effect and then applies the other function. When this function returns, the closure bindings are removed. Closures come in two kinds: dynamic and lexical. Dynamic closures are made with the function **closure**, and lexical closures are made with the special form **function**, **flet**, or **labels**. See Section 17, Closures, for details.

A flavor instance is a message-receiving object that has both a state and a table of message-handling functions (called *methods*). Refer to Section 19, Flavors, for further information.

An array can be used as a function. The arguments to the array are the indices, and the value is the contents of the element of the array. This arrangement is for MacLisp compatibility and is not recommended usage. Use **aref** instead.

A stack group can be called as a function. This is one way to pass control to another stack group (see Section 26, Stack Groups).

Defining Functions

16.3 The following macros and special forms are used to define named functions.

defun *name lambda-list {declaration|doc-string}* {body-form}** *[c]* Macro

This macro is the usual way of defining a function that is part of a program.

The *name* argument specifies the function spec you want to define as a function. The *lambda-list* has the same meaning as a lambda list in a lambda expression.

The **defun** macro creates a list with the following format:

(**named-lambda** *name lambda-list {body-form}**)

This list is put in the function spec location of *name*. The *name* argument is now defined as a function and can be called by other forms. When the function is called, the *body-forms* are executed in sequence, and the value of the last form is returned as the value of the function call. Alternatively, when *name* is a symbol, the form (**return-from** *name value*) can be used to exit the function in the same manner as a **block** form.

Declarations (lists starting with the symbol **declare**) can appear as the first elements of the body. For details see Section 13, Declarations.

A *doc-string* can also appear as the first element of the body either before or after the declarations, if there are any. It should not be the only element in the body; otherwise, it is the value returned by the function and thus is not interpreted as documentation. This documentation string becomes part of the function's debugging information and can be obtained with the function **documentation**. The **documentation** function is also a suitable *place* argument for **setf** to update a *doc-string*.

There are two levels of documentation carried with the definition: the lambda list and the explicitly supplied documentation. If, in the Lisp Listener or the Zmacs editor, you type an opening parenthesis and a symbol that has a function definition and then press CTRL-SHIFT-A, the lambda list is printed on the screen. If you press CTRL-SHIFT-D, the lambda list is printed along with the explicit documentation supplied with the **defun**. For this reason, it is particularly useful for you to use meaningful parameter names and defaults. If you expect this form to be invoked by a user at the top level, then the knowledgeable user should be able to understand how to use the form simply by reading the argument list and the documentation string.

Usually a **defun** is used as a top-level form, but Common Lisp also permits it to be used within the lexical context of forms such as **let**, **flet**, and **macrolet**. In such a case, the **defun** still defines the function to be globally accessible, but execution of the function has access to the lexical environment of the definition. For example:

```
(let ((count 0))
   (defun counter ()
      (setq count (+ count 1))
      count))
```

This example defines a function that returns the count of the number of times it has been called. Since the variable `count` is bound outside the `defun`, it retains its value between calls to the function. But since it is not a special variable, it cannot be accessed by other functions. Note that this usage provides a capability equivalent to what is called *local static data* in some other programming languages.

defsubst *name lambda-list* {*body-form*}* Macro

This macro is used for defining substitutable functions. It is used just like **defun** and performs almost the same operation. It defines a function that executes identically to the one that a similar call to **defun** would define. The difference between **defun** and **defsubst** is that when a function that calls **defsubst** is compiled, the call is open-coded by inserting the substitutable function's definition into the code being compiled. As with **defun**, *name* can be any function spec. The function itself looks like the form `(named-subst name lambda-list . body)`. Such a function is called a *subst*. For example:

```
(defsubst square (x) (* x x))
(defun foo (a b) (square (+ a b)))
```

In this example, if `foo` is used interpreted, then `square` works just as if it had been defined by **defun**. If `foo` is compiled, however, the squaring is substituted into `foo` to produce the same code as the following:

```
(defun foo (a b) (let ((tem (+ a b))) (* tem tem)))
```

Thus, `square`'s definition would be the following:

```
(named-subst square (x) (* x x))
```

For more information on substs, see paragraph 16.2.1, Interpreted Functions.

A similar `square` could be defined as a macro:

```
(defmacro square (x)
   (once-only (x)
      `(* ,x ,x)))
```

In general, anything that is implemented as a subst can be reimplemented as a macro just by changing the **defsubst** to a **defmacro**, putting in the appropriate backquote and commas, and using **once-only** or creating temporary variables to ensure that the arguments are computed once and in the proper order. The disadvantage of macros is that they are not functions and thus cannot be applied to arguments. Another disadvantage is the effort required to guarantee the order of evaluation. The advantage of macros is that they are much more powerful than substs. However, this is also a disadvantage because macros provide more ways to get into trouble. If something can be implemented either as a macro or a subst, it is generally better to make it a subst.

The *lambda-list* of a subst can contain &optional and &rest lambda-list keywords but no other lambda-list keywords. If there is a &rest argument, it is replaced in the body with an explicit call to **list**:

```
(defsubst append-to-foo (&rest args)
   (setf foo (append args foo)))
```

```
(append-to-foo x y z)
```

The preceding form expands to the following:

```
(setq foo (append (list x y z) foo))
```

Any &rest arguments in substs are most useful with **apply**. For example:

```
(defsubst xhack (&rest indices)
   (apply 'xfun xarg1 indices))
```

Because of an optimization, if the preceding form has been executed, then the following equivalence is true:

```
(xhack a (car b)) <=> (xfun xarg1 a (car b))
```

If `xfun` is itself a subst, it is expanded in turn.

When a **defsubst** is compiled, its list structure definition is kept so that calls can still be open-coded by the compiler. But non-open-coded calls to the function run at the speed of compiled code. The interpreted function is kept in the compiled definition's debugging info association list. Undeclared free variables used in a **defsubst** that is being compiled do not produce any warning because this is a common practice that works properly with nonspecial variables when calls are open-coded.

If you are using a **defsubst** from outside the program to which it belongs, you may be better off if it is not open-coded. The decrease in speed may not be significant, and you would not need to recompile your program if the definition changes. You can prevent open-coding by putting **dont-optimize** around the call to the **defsubst**:

```
(dont-optimize (xhack a (car b)))
```

Straightforward substitution of the argument could cause arguments to be computed more than once or in the wrong order. For example:

```
(defsubst reverse-cons (x y) (cons y x))
(defsubst in-order (a b c) (and (< a b) (< b c)))
```

The preceding functions would cause problems. Because of the substitution, a call to `reverse-cons`, when compiled, would evaluate its arguments in the wrong order, and a call to `in-order` could evaluate its second argument twice. In fact, a more complicated form of substitution is used so that local variables are introduced as necessary to prevent such problems.

Note that all occurrences of the argument names in the body are replaced with the argument forms, wherever they appear. Thus, an argument should not be used in the body for anything else, such as a function name or a symbol in a constant.

Other Function-Defining Forms

16.4 The following are several function-defining special forms with more restricted uses than **defun** has.

def *function-spec {form}** Macro

If a function is created in an unusual way, you can wrap this macro around the code that creates this function to inform the editor of the connection. The *function-spec* argument is not evaluated. This form simply evaluates the *forms*. It is assumed that these forms create or obtain a function in some way and make this function the definition of *function-spec*.

Alternatively, you can put (**def** *function-spec*) in front of or anywhere near the forms that define the function. The editor uses it only to tell on which line to put the cursor.

deff *function-spec definition-creator* Macro

This macro evaluates the form *definition-creator*, which should produce a function, and makes that function the definition of *function-spec*, which is not evaluated. The **deff** macro is used for giving a function spec a definition not obtainable with the specific defining forms such as **defun**. For example:

```
(deff foo 'bar)
```

The above form makes `foo` synonymous with the symbol `bar`, (an indirection so that if `bar` changes, `foo` likewise changes).

The following form copies the definition of `bar` into `foo` with no indirection so that further changes to `bar` have no effect on `foo`:

```
(deff foo #'bar)
```

deff-macro *function-spec definition-creator* Macro

This macro is like **deff** but is used for defining macros. The **definition-creator** argument is evaluated to produce a definition suitable as a macro, and then *function-spec* is defined as a macro. The macro definition should be a cons whose car is **macro** and whose cdr is an expander function. Alternatively, a substitute function definition can be used, either a list starting with **subst** or **named-subst**, or a compiled function that records that it was compiled from such a list. (See paragraph 16.2.2, Compiled Functions.)

The difference between **deff** and **deff-macro** is that **compile-file** assumes that **deff-macro** is defining something that should be expanded during compilation. For the rest of the file, the macro defined by **deff-macro** is available for expansion. When the file is ultimately loaded or if compilation is done in memory from the Lisp Listener using **compile**, **deff** and **deff-macro** are equivalent.

Passing and Receiving Multiple Values

16.5 Most Lisp functions return a single value. In cases when a function wants to return several values, it could cons up a list to be returned. Of course, this procedure is a waste of time and memory if the receiving function is going to rebind each of these values to a local variable. An economical solution to this problem is to let the called function return multiple values and let the calling function decide how the various returned values should be handled. No error is produced if multiple values are sent to the caller of a function that does not have the ability to receive multiple values. This caller simply takes the first value of the returned values and discards the rest.

values &rest *values* *[c]* Function

This function returns multiple values: its arguments. This function is the primitive function for producing multiple values. For example:

```
(defun pushnew-member-p (value place)
   (let ((member-p (member value place)))
      (values (pushnew value place) member-p)))
```

If **value** is a member of **place**, this example returns a second value which is true. Note that if you define a function that is to produce multiple values, the **values** function must be the last form evaluated. With no arguments, **values** produces no return values. For example:

```
(length (multiple-value-list (values))) => 0
```

When functions only have side effects (such as printing a message), it may be desirable to return no values.

multiple-values-limit *[c]* Constant

This constant contains a positive integer indicating the limit for the number of values any form can return. This integer is an upper exclusive bound; therefore, since the value of **multiple-values-limit** on the Explorer system is 64, a form can return up to 63 values.

values-list *list* *[c]* Function

This function returns the elements of *list* as multiple values. The argument *list* can be **nil**, the empty list, which causes no values to be returned. Note the following equivalence:

```
(values-list '(x y z)) <=> (values 'x 'y 'z)
(values-list list)     <=> (apply #'values list)
```

multiple-value-bind ({*var*}*) *form* {*declaration*}* {*body-form*}* *[c]* Special Form

This special form first evaluates *form* and binds the *var* variables to the values returned by *form*. The *body-forms* are then evaluated sequentially and the result of the last *body-form* is returned. If more values are returned than there are variables, the extra values are ignored. If there are more variables than values returned, extra values of **nil** are supplied. For example, the **find-symbol** function returns three values:

```
(multiple-value-bind (symbol status actual-package)
            (find-symbol "CONS")
   (format nil "The symbol:~a, Status:~a, Package:~a"
          symbol status actual-package))
=> "The symbol:CONS, Status:INHERITED, Package:LISP"
```

multiple-value-setq ({*var*}*) *form* *[c]* Special Form

This special form is used for calling a function that is expected to return more than one value. The argument *form* is evaluated, and the *var* variables are *assigned,* not bound, to the values returned by *form*. If more values are returned than there are variables, the extra values are ignored. If there are more variables than values returned, extra values of **nil** are supplied. For example:

```
(multiple-value-setq (symbol already-there-p) (intern "goo"))
```

In addition to its first value, the symbol, `intern` returns a second value, which is **true** if an existing symbol was found or else **nil** if `intern` had to create one. So if the symbol `goo` was already known, the variable `already-there-p` is set to a **true** value; otherwise, it is set to **nil**. The third value returned by `intern` is ignored by this example because there is no third variable in the **multiple-value-setq**.

The **multiple-value-setq** special form is usually used for effect rather than for value; however, its value is defined to be the first of the values returned by *form*.

As an extension on the Explorer system when **nil** appears in the *var* list the corresponding value is ignored.

multiple-value-list *form* *[c]* Special Form

This special form evaluates *form* and returns a list of the values returned from this evaluation. For example:

```
(multiple-value-list (floor 11. 7.)) => (1 4)
```

This special form is useful when you do not know how many values to expect.

multiple-value-call *function* {*form*}* *[c]* Special Form

This special form evaluates the *forms*, saving all of their values, and then calls *function* with all those values as arguments. This differs from (**funcall** *function argforms* ...) because the **funcall** form receives only one argument for *function* from each *argform*, whereas **multiple-value-call** receives as many arguments from each *form* as the *forms* return. This works by concatenating a list of all the values returned and applying *function* to it. For example:

```
(multiple-value-call
  #'list (values 1 2) (values) (values 3))
=> (1 2 3)
```

multiple-value-prog1 *form1* {*form*}* *[c]* Special Form

This special form evaluates *form1*, saves its values, evaluates the *forms*, discards their values, and returns whatever values *form1* produced.

nth-value *n form* Special Form

This special form evaluates *form* and returns its *n*th value, where *n* = 0 means the first value. For example, `(nth-value 1 (foo))` returns the second of `foo`'s values. This special form operates without consing in compiled code if *n*'s value is known at compile time.

Rules for Passing Multiple Values

16.6 The following are rules that the system uses for functions that pass and receive multiple values. Note that if a form is used to generate an argument to a function, that function receives only the first value returned by the form, even if the form returns multiple values.

Evaluation and Application

- The **eval** function returns multiple values only when the form that it evaluates returns multiple values.

- The forms **apply**, **funcall**, and **multiple-value-call** return multiple values if the function they apply returns multiple values.

progn *and* **progn**-*Like Constructs*

- The **progn** special form passes back multiple values if its last subform returns multiple values. Other forms that follow this rule include the following:

  eval-when
  unless
  typecase
  progv
  block
  ecase
  let
  multiple-value-bind
  etypecase
  let*
  ctypecase
  when
  catch
  ccase

  Forms created by the following also follow this rule:

  lambda
  defun
  defmacro
  deftype

Conditional Constructs

- The **if** special form returns multiple values if the selected form (the *then* clause or the *else* clause) returns multiple values.

- The **and** and **or** special forms return multiple values if the last subform (that is, the last argument to the **or** or **and**) is evaluated and if this subform returns multiple values.

- The **cond** macro returns multiple values if the last subform of the selected clause does, unless the clause consists of only a test form.

Block Constructs

- The **block** special form returns multiple values if its last subform does. However, if a **block** is terminated by a **return-from** or **return** form, then it passes back multiple values only if the return forms do. Other forms that follow this rule include **do**, **dolist**, **dotimes**, **prog**, and **prog***.

- The **do** macro returns multiple values if the last subform of the exit clause does. The **dolist** and **dotimes** macros return multiple values if their result forms do.

- The **catch** special form returns multiple values if the last form returns multiple values or if the result of the **throw** generates multiple values.

Miscellaneous Constructs

- The **multiple-value-prog1** special form returns multiple values if its first subform does, whereas the **prog1** macro passes back a single value in all cases.

- The **unwind-protect** special form returns multiple values if its protected form does.

- The **the** special form returns multiple values if the form that it evaluates does.

Forms That Never Return Multiple Values

- The following forms never return multiple values: **setq**, **prog1**, **prog2**, and **multiple-value-setq**.

Evaluation and Application Forms

16.7 *Evaluation* is the process of extracting value from any Lisp object. *Application* is the process of binding argument values to a function's parameters and executing the function with those bindings. The evaluation process uses application in that a function must first be applied to argument values to evaluate or extract a value from that function.

Normally, evaluation and application are implicitly invoked whenever a form is executed, but they can be explicitly invoked by using the **eval** or **apply** functions. It is also possible to inhibit evaluation and application by the **quote** and **function** special forms. This ability to inhibit evaluation and application allows Lisp to have the very powerful characteristic that data and executable code can be handled the same way.

Thus, symbols can have not only their function definition cells set to functional objects but also their value cells. This assignment can be performed using the **function** special form in combination with setting and binding operations. A symbol that has executable code as its value can be manipulated in any manner appropriate for a symbol whose value is simply a data object. This flexibility allows functional definitions and function call forms to be manipulated as data until a point of explicit application or evaluation is encountered.

Explicit Evaluation

16.7.1 The following functions are associated with explicit evaluation.

eval *form* &optional *environment* *[c]* Function

This function evaluates *form* and returns the result. For example:

```
(defparameter x 0)

(eval (list '+ 'x 1)) => 1
```

Note that the argument passed to `eval` is the form `(+ x 1)`. Remember that the **eval** function recognizes only dynamic bindings:

```
(defparameter x 0)

(let ((x 1))
   (values x (eval 'x))) => 1 1
```

The `defparameter` provides a global value for `x`, but more interesting in this case is that it also proclaims `x` to be special. When the `let` form is executed, `x` receives a dynamic binding. The argument that is passed to `eval` is the symbol `x`, which at this point is set to `1`. Consider the following counterexample:

```
(setq non-special-x 0)

(let ((non-special-x 1))
   (values non-special-x
          (eval 'non-special-x))) => 1 0
```

Note that on the Explorer system simply executing a `setq` at the top level does not proclaim a special variable, and thus the `let` does not produce a dynamic binding; consequently, `eval` sees the original global binding. The following example is another case of the same problem:

```
(let ((a-local-lexical-var 0))
   (eval 'a-local-lexical-var)) => ERROR
```

Remember that, like all other functions, **eval** cannot see lexical bindings defined in the calling function.

It is unusual to call **eval** explicitly, because evaluation is usually done implicitly. If you are writing a simple Lisp program and explicitly calling **eval**, you are probably doing something wrong.

Also, if you are interested only in retrieving the dynamic value of a symbol (that is, the contents of the symbol's value cell), then you should use the primitive function **symbol-value.**

The *environment* argument is a lexical environment that includes local variables and macro definitions. This environment is bound to a variable using the &environment lambda-list keyword in a macro. This feature is useful only if you are supplying an ***evalhook*** function, in which case you will receive a form and an environment. Eventually, your ***evalhook*** function will probably want to call **eval** to process the remainder of the form, at which point you should supply the environment argument to **eval**. For more information on ***evalhook***, see the *Explorer Tools and Utilities* manual.

NOTE: **eval** may alter its argument so that subsequent evaluations will be faster. See Paragraph 18.5, Displacing Macro Calls, for more information.

---

sys:*eval *form* Function

This function evaluates *form* in the interpreter's *current* lexical environment. This function is used by special forms to evaluate their arguments.

Explicit Application

16.7.2 Although most functions are executed implicitly through evaluation, in some cases they must be explicitly applied, for instance, whenever a function resides in the value cell of a symbol. Also, any function spec that is not a symbol must be explicitly applied. The following functions are associated with explicit application.

apply *fn* {*arg*}* *arglist* *[c]* Function

This function applies the function *fn* to a list of arguments. Its main purpose is to enable, at run time, the calling of an arbitrary function with an arbitrary number of arguments. This is an important feature because many functions have an optional (varying) number of parameters.

The argument list is formed by logically consing each *arg* onto *arglist* to make a list that maintains the order in which the arguments were supplied. The *fn* argument can be any function, but it cannot be a macro or special form. If *fn* is a symbol, then the global **symbol-function** value is used.

The *arglist* must be a list, which can be **nil**. If *fn* is not supposed to receive any arguments, then do not supply any *args* and specify *arglist* as **nil**. For example:

```
(apply '+ 1 1 1 '(1 1 1)) => 6

(defun report-error (&rest args)
     (apply #'format *error-output* args))
```

funcall *fn* &rest *args* *[c]* Function

This function applies the function *fn* to the arguments *args*. It is similar to **apply** except that with **funcall** you must know the number of arguments that the function is being applied to. For **funcall**, if the last *arg* is a list, then that list is assigned as the value of the corresponding parameter in *fn*.

The functional argument *fn* cannot be a special form or a macro; this would not be meaningful because arguments would probably be evaluated incorrectly. If *fn* is a symbol, then the global **symbol-function** value is used. For example:

```
(cons 1 2) => (1 . 2)          ; Implicitly invoke the cons function.
(setq cons '+)                 ; Set the values cell of the cons symbol.
(funcall cons 1 2) => 3        ; Access the value cell of cons.
(funcall 'cons 1 2) => (1 . 2) ; Access the function cell of cons.
```

This example shows that the use of the symbol `cons` as the name of a function and the use of that symbol as the name of a variable do not interact. The `cons` form invokes the function named `cons`. The first `funcall` form evaluates the variable and receives the symbol +, which is the name of a different function. The second `funcall` form receives the `cons` symbol as an argument.

In **funcall**, the arguments to be applied are given as individual arguments to *fn*. Thus, **apply** is useful for passing a varying number of arguments, while **funcall** is usually more convenient for passing predetermined arguments:

```
(funcall 'fn a b) <=> (fn a b)
```

In this example, `fn` is a symbol.

send *object operation* &rest *arguments* Macro

This macro sends *object* a message with *operation* and *arguments* as specified. This macro is equivalent to the following but is more meaningful for object-oriented code:

```
(send object operation arg1 arg2 ...argn) <=>
(funcall object operation arg1 arg2 ...argn)
```

lexpr-send *object operation* {*arg*}* *arglist* Macro

Currently, this macro is equivalent to the following but is more meaningful for object-oriented code:

```
(lexpr-send object operation arg1 arg2 ...argn arglist) <=>
(apply object operation arg1 arg2 ...argn arglist)
```

call *fn* &rest *key-vals* Function

This function offers a very general way of controlling what arguments you pass to a function. You can provide either individual arguments, like **funcall**, or lists of arguments, like **apply**, in any order. In addition, you can make some of the arguments optional. If the function is not prepared to accept all the arguments you specify, no error occurs if the excess arguments are optional ones. Instead, the excess arguments are simply not passed to the function. The *key-vals* argument specifies alternating keyword delimiters and values. Each delimiter indicates what to do with the value that follows.

The following are the four acceptable delimiters:

- **:optional** — All of the remaining arguments to the right are optional arguments for *fn*.
- **:spread** — The following value is a list, and each element in the list should be supplied as a separate argument to *fn*.
- **'(:optional :spread)** — The value is both **:optional** and **:spread**.
- () — The value requires no delimiter. In practice, this delimiter specifies to not spread the next argument. Also, this argument is optional if the **:optional** keyword appears previously in the form.

Consider the following example:

```
(call #'foo () x :spread y '(:optional :spread) z () w)
```

The arguments passed to `foo` are the value of `x`, the elements of the value of `y`, the elements of the value of `z`, and the value of `w`. The function `foo` must be prepared to accept all the arguments that come from `x` and `y`, but if it does not want the rest, they are ignored.

call-arguments-limit *[c]* Constant

This constant has as its value an integer specifying the limit for the number of arguments that can be handled by a function call. This integer is an upper exclusive bound; therefore, since the value for **call-arguments-limit** is 64 on the Explorer system, each function call can handle up to 63 arguments.

The following three functions are useful to pass as arguments to functions that are to subsequently **funcall** or **apply** these arguments.

ignore &rest *objects* Function

This function takes any number of arguments and returns **nil**. This function is often used as a dummy function. Note that this form is different from the declaration specifier **ignore** (see paragraph 13.3, Declaration Specifiers) although this form is often used for the same purpose.

identity *object* *[c]* Function

This function returns the value of *object*. If *object* is a form that produces multiple values, only the first value is returned.

true Function

This function takes no arguments and returns **t**.

false Function

This function takes no arguments and returns **nil**.

Evaluation Inhibition and the function Form

16.7.3 The following two functions are used to inhibit evaluation and application.

quote *object* *[c]* Special Form

This special form inhibits the process of evaluation when this form surrounds an object. The form (**quote** *object*) simply returns *object*. The **quote** special form is used to include constants in a form. It is useful specifically because *object* is not evaluated. For example:

```
(quote x) => x
(setf x (quote (some list)))
x => (some list)
```

The form (**quote** *object*) can be abbreviated as '*object*. The Lisp Reader normally converts any form preceded by a single quote (') character into a quote form. For example:

```
(setf x '(some list))
```

This form is converted by **read** into the following:

```
(setf x (quote (some list)))
```

function *fn* *[c]* Special Form

This special form has two distinct, though related, meanings.

The first meaning deals with *fn* when it is a function spec. If *fn* is a symbol or any other function spec, the form (**function** *fn*) returns the function definition of *fn*. For example, in `(mapcar (function car) x)`, the function definition of `car` is passed as the first argument to `mapcar`. The form `function` used in this way is like **fdefinition**, except that its argument is unevaluated, so `(function fred)` is like `(symbol-function 'fred)`. Also, **function** can reference locally defined functions (see **flet**), whereas **fdefinition** and **symbol-function** access only globally defined functions.

The second meaning deals with *fn* when it is a lambda expression. In this case, (**function** *fn*) represents that function suitably interfaced to execute in the lexical environment where it appears. For example:

```
(let (y)
  (mapcar (function (lambda (x)
                      (push x y)))
          a-list)))
```

This `let` form passes `mapcar` a specially designed closure (for a discussion of lexical closures, see Section 17, Closures) made from the function definition represented by `(lambda (x) (push x y))`. When `mapcar` calls this closure, the lexical environment of the function form is put into effect again, and the `y` in `(push x y)` refers properly to the binding made by this `let`. Additionally, the compiler knows that the argument to **function** should be compiled.

To make typing easier, the Lisp Reader converts *#'thing* into (**function** *thing*). The *#'* form is similar to ', except that it produces a **function** form instead of a **quote** form. (The argument of **quote** cannot be compiled because it may be intended for other uses.) Thus, the previous example could be written as the following:

```
(let (y)
  (mapcar #'(lambda (x)
              (push x y))
          a-list))
```

Another way of explaining **function** is that it causes *fn* to be treated the same way it would be as the car of a form being evaluated. Evaluating the form *(fn arg1 arg2...)* uses the function definition of *fn* if *fn* is a function spec; otherwise, *fn* is expected to be a list that is a lambda expression.

You should be careful about whether you use *#'* or '. Suppose you have a program with a variable `x` whose value is assumed to contain a function that is called on several arguments. If you want that variable to be the `test` function, there are two forms you could use:

```
(setf x 'test)
```

```
(setf x #'test)
```

The former causes the value of `x` to be the symbol `test`, whereas the latter causes the value of `x` to be the function object found in the function cell of `test` at the time the `setf` was executed. When the time comes to call the function (the program executes `(funcall x...))`, either expression works because calling a symbol as a function uses its function definition instead. Using `'test` is insignificantly slower, because the function call has to go indirectly through the symbol, but it allows the function to be redefined, traced, or advised (see the *Explorer Tools and Utilities* manual for Advising a Function). The latter case, while faster, picks up the function definition out of the symbol `test` when the `setf` is executed and does not see any later changes to it.

Functions That Manipulate Function Specs

16.8 The following functions and variables can be used to manipulate function specs.

fdefine *function-spec definition* &optional *carefully-p no-query-p* Function

This function is the primitive used by **defun** and everything else in the system to change the definition of a function spec. If *carefully-p* is non-**nil**, which it usually should be, then only the basic definition is changed; the previous basic definition is saved if possible (see **undefun** later in this section), and any encapsulations of the function, such as tracing and advice, are carried over from the old definition to the new definition. The *carefully-p* argument also causes the user to be queried if the function spec is being redefined by a file different from the one that defined it originally. However, this warning is suppressed if either the argument *no-query-p* is true or if the global variable **inhibit-fdefine-warnings** is true.

If **fdefine** is called while a file is being loaded, it records which file the function definition came from so that the editor can find the source code.

If *function-spec* was already defined as a function and *carefully-p* is true, the function-spec's **:previous-definition** property is used to save the previous definition. This property is used by the **undefun** function, which restores the previous definition. The properties for different kinds of function specs are stored in different places; when a function spec is a symbol, its properties are stored on the symbol's property list.

The **defun** macro and the other function-defining special forms all supply true for *carefully-p* and **nil** or nothing for *no-query-p*. Operations that construct encapsulations, such as **trace**, are the only ones that use **nil** for *carefully-p*.

sys:**record-source-file-name** *name* &optional *type no-query-p* — Function

This function records a definition of *name*, of the type specified by *type*. The *type* argument should be a symbol such as **defun** to record a function definition; if it is, *name* is a function spec. The *type* argument can also be **defvar**, **defflavor**, **defresource**, **defsignal**, or anything else you want to use.

The value of **sys:fdefine-file-pathname** is assumed to be the generic pathname of the file that the definition is coming from, or **nil** if the definition is not from a file. If a definition of the same *name* and *type* has already been seen but not in the same file, and *no-query-p* is **nil**, then an error condition is signaled and the user is queried.

If **sys:record-source-file-name** returns **nil**, it means that the user or a condition handler specified that the redefinition should not be performed.

sys:**fdefine-file-pathname** — Variable

While the system is loading a file, this variable specifies the generic pathname for the file. The rest of the time it is **nil**. The **fdefine** function uses this variable to remember which file defines each function.

sys:**get-source-file-name** *name* &optional *type* — Function

This function returns the generic pathname for the file in which *name* received a definition of the type specified by *type*. If *type* is **nil**, the most recent definition is used, regardless of its type. The *name* argument is a function spec if *type* is **defun**; if *type* is **defvar**, *name* is a variable name. Other types that are used by the system are **defflavor**, **defstruct**, **defparameter**, **defresource**, **defsignal**, and so on.

This function returns the generic pathname of the source file. To obtain the actual source file pathname, use the **:source-pathname** operation on the generic pathname object.

A second value is returned, which is the type of the definition that was reported.

sys:**get-all-source-file-names** *function-spec* — Function

This function returns a list describing the generic pathnames of all the definitions this function spec has received, of all types. The list is an association list whose elements have the form (*type pathname*...).

inhibit-fdefine-warnings Variable

This variable is normally **nil**. Setting it to **t** prevents **sys:record-source-file-name** from warning you and asking about questionable redefinitions, such as a function being redefined by a different file than defined it originally, or a symbol that belongs to one package being defined by a file that belongs to a different package. Setting this variable to **:just-warn** allows the warnings to be printed out but prevents the queries from happening; it assumes that your answer is yes—that is, that the function can be redefined.

sys:validate-function-spec *object* Function

This predicate returns true if the argument is a legal function spec for the Explorer system. Recall that Common Lisp only allows symbols for function specs.

fdefinedp *function-spec* Function

This function returns true if *function-spec* has a definition, or **nil** if it does not.

fdefinition *function-spec* Function

This function returns the definition of *function-spec*. If it has none, an error occurs. This function is similar to the special form **function** but differs in that the argument is evaluated. As with **function**, if the argument is a symbol, the function cell is returned. Thus, **function** can be used when the function spec is a constant, but **fdefinition** is needed when the function spec is to be a variable at run time.

fundefine *function-spec* Function

This function makes *function-spec* undefined; the cell where its definition is stored becomes void. For symbols, this function is equivalent to **fmakunbound**. If the function is encapsulated, **fundefine** removes both the basic definition and the encapsulations. Some types of function specs (**:location**, for example) do not implement **fundefine**. Invoking **fundefine** on a **:within** function spec removes the replacement of *function-to-affect*, putting the definition of *within-function* back to its normal state. Invoking **fundefine** on a **:method** function spec removes the method completely so that future messages are handled by another method (see Section 19, Flavors).

undefun *function-spec* Function

If *function-spec* has saved a previous basic definition, this function interchanges the current and previous basic definitions, leaving the encapsulations alone. If *function-spec* has no saved previous definition, **undefun** asks the user whether to make it undefined.

This function cancels the effect of redefining a function. See also **uncompile** in paragraph 21.2, Invoking the Compiler.

sys:function-spec-get *function-spec indicator* Function

This function returns the value of the *indicator* property of *function-spec*, or **nil** if it has no such property.

sys:function-spec-putprop *function-spec value indicator* Function

This function gives *function-spec* an *indicator* property whose value is *value*.

sys:function-spec-lessp *function-spec1 function-spec2* Function

This function compares the two function specs specified by *function-spec1* and *function-spec2* with an ordering that is useful in sorting lists of function specs for presentation to the user.

sys:function-parent *function-spec* Function

If *function-spec* does not have its own definition, textually speaking, but is defined as part of the definition of another object, this function returns the function spec for that other object. For example, if *function-spec* is an accessor function for a **defstruct**, the value returned is the name of the **defstruct**.

The reason for this procedure is that if the caller has not been able to find the definition of *function-spec* in a more direct fashion, it can try looking for the definition of the function-parent of *function-spec*. The function-parent of a function spec is defined by including a declaration of **sys:function-parent** in the function spec. See Section 13, Declarations, for details.

This function helps the editor find the proper source file definition for a function.

Defining Local Functions

16.9 The following forms allow you to define a named local function within the scope of another function.

flet ({(*fn-name lambda-list* {*declaration* | *doc-string*}* { *fn-name-form*}*)}*) {*declaration*}* {*flet-body-form*}* *[c]* Special Form

This special form is a **let** for functions, allowing you to define functions (rather than variables as with **let**) local to the **flet** body. The local function definitions called *fn-name* take the same appearance as a **defun** in that they contain a lambda list, optional declarations and a documentation string, and a function body. The local functions can only be called within the lexical scope of the **flet** body. For example:

```
(defun foo (x)
;; The function "foo" uses two local functions.
 (flet ((bar (y)
             (+ 1 y))
        (baz (z)
             (- 2 z)))
    ;; The body of flet calls both functions.
  (cond ((= x 0) (bar x))
         (t (+ (bar x) (baz x))))))

(foo 0) => 1
(foo 3) => 3
```

Each local function is closed in the environment outside the `flet`. As a result, the local functions cannot call each other or recurse. They can, of course, call global functions. In the above example, the function `foo` implements a `flet` form that defines two local functions—`bar` and `baz`. These functions are then used in the `cond` form of the body of the `flet`.

To allow local functions to call each other and recurse, use labels.

labels (*{(fn-name lambda-list {declaration | doc-string}* {fn-name-body}*)}**) *{declaration}* {labels-body}** *[c]* Special Form

This special form is like **flet**, except that the local functions can call each other and recurse. They are closed in the environment inside the **labels**, so all the local function names are accessible inside the bodies of the local functions. For example:

```
(defun entropy (prbs) ; An obscure function.
 ;; Entropy has 2 local functions "sum" and "ln".
 (labels ((sum (lst)
             (cond ((null lst) 0)
                       ;; sum calls "ln" and also recurses.
                   (t (+ (* (car lst) (ln (car lst)))
                         (sum (cdr lst))))))
          (ln (x)
             (log (/ 1 x))))
       ;; Call and return value of local function "sum".
   (sum prbs)))
```

The **labels** special form is one of the most ancient Lisp constructs but was typically not implemented in second generation Lisp systems in which no efficient form of closures existed. See also **macrolet**, an analogous construct for defining macros locally, which is described in paragraph 18.4, Local Macro Definitions.

Special Forms

16.10 The special forms, such as **quote** and **let**, are actually implemented with an unusual sort of function. This paragraph explains how the evaluator handles special forms.

Recall that when the evaluator is given a list whose first element is a symbol, the form can be a function form, a special form, or a macro form. If the function definition cell of the symbol is a function, then the function is simply applied to the result of evaluating the rest of the argument subforms. If the definition is a cons whose car is **macro**, then it is a macro form; macros are explained in Section 18, Macros.

A special form is implemented by a function that is flagged to tell the evaluator to refrain from evaluating some or all of the arguments to the function. Such functions make use of the lambda-list keyword &quote.

The **eval** function, on seeing the &quote in the lambda list of an interpreted function (or the equivalent in a compiled function), skips the evaluation of the arguments in the call form that correspond to &quote parameters in the lambda list of the interpreted function. Aside from that, it calls the function normally.

For example, **quote** can be defined as follows:

```
(defun quote (&quote arg) arg)
```

Evaluation of `(quote x)` recognizes the `&quote` in the definition and recognizes that it refers to the first argument, so it refrains from evaluating that argument and passes to the function definition of `quote` the object `x` rather than the value of `x`. From then on, the definition of `quote` executes in the normal fashion, so it returns `x`.

The &quote lambda-list keyword has this effect on all the subsequent parameters, but it can be canceled with &eval. A simple **setq** that accepts only one special variable and one value can be defined as follows:

```
(defun setq (&quote variable &eval value)
 (set variable value))
```

The actual definition of **setq** is more complicated and uses a lambda list `(&quote &rest variables-and-values)`. Then it must go through the &rest argument, evaluating every other element.

The definitions of special forms are designed with the assumption that they will be called by **eval**. It does not usually make much sense to call one with **funcall** or **apply**.

You can define your own special form using &quote. The reasons for doing so are limited, however. The simplest case is one in which the argument being passed is always treated as a constant within your function. The &quote is sometimes used as a matter of programming convenience, so the caller does not have to quote (') the argument; this convenience is best reserved for forms that are typed from the Lisp Listener. If a &quoted parameter is eventually to be evaled, the risk of undesired results is great. Consider the following example:

```
(defun test (&quote thing)
     (eval thing))

(let ((x 0))
   (test x)) => ERROR
```

In this example, the variable `x` only has a defined value within the lexical scope of the `let` statement. However, the `test` function body (which is outside the lexical scope of the `let`) is being asked to `eval x`; therefore, `x` is a reference to an unbound variable in `test`. The variable `x` could have been declared special within the `let` to solve this particular problem, but you also might want the `test` function to declare its own special variable `x`. You can use the **sys:*eval** function to solve these problems. Macros avoid these problems by using textual substitution within the lexical scope of the calling function.

How Programs Examine Functions

16.11 The following functions take a function as an argument and return information about the function. Some also accept a function spec and operate on its definition. The others do not accept function specs in general but do accept a symbol as standing for its definition.

sys:get-debug-info-struct *function* &optional *unencapsulated-p* Function

This function returns a structure of type **debug-info-struct** that describes information relevant to *function*. Use this returned value as an argument to the **sys:get-debug-info-field** function or send the **:describe** message to the structure to get a full description of *function*. The *function* argument can be a function spec or function object. If *unencapsulated-p* is true, then the returned value is the debug-info structure for the base function. Encapsulations are described in paragraph 16.12, Encapsulations.

sys:get-debug-info-field *debug-info-structure field-name* Function

This function is used to extract information from a debug-info structure. The *debug-info-structure* argument should be an object of type **debug-info-struct**. For a particular function, this structure is made accessible by using the **sys:get-debug-info-struct** function. The *field-name* argument should be a keyword that describes a particular attribute in which you are interested. The following are some of the more useful attributes:

- **:name** — Returns the name of the function.
- **:arglist** — Returns the argument list for the function.
- **:interpreted-definition** — Returns the interpreted version of this function if it is available. If the function was proclaimed **inline** or is a **defsubst**, then this information should be available.
- **:local-map** — Returns the layout of the local variables used by the function.
- **:plist** — Returns a property list of other attributes. The property list often contains some of the following:
    - **:macros-expanded** — A list of macros that were expanded when the function was compiled.
    - **:documentation** — The function's documentation string.
    - **:descriptive-arglist** — The argument list for the function as defined by the form (**declare** (**arglist** . . .)). Because this argument is purely for documentation and need not even exist, it is not used by the compiler or the interpreter.
    - **:values** — The returned list of values as defined by the form (**declare** (**values** . . .)).
    - **:internal-fef-offsets** — Describes the addresses within the FEF of the function cells for internal functions of the FEF.
    - **:internal-fef-names** — A list of the names of the internal functions of the FEF.
    - **:function-parent** — Specifies the name of a definition whose source code includes this function. This attribute is for functions defined automatically by **defstruct**, **defflavor**, and so on.
    - **sys:encapsulated-definition** *internal-symbol type-of-encapsulation* — This attribute means that this function was made to encapsulate an inner definition.
    - **sys:renamings** *alist-of-renamings* — This item is used together with the form (**encapsulated-definition** . . . **:rename-within**) and specifies what renamings are to be performed for the original definition. Each element of the association list has the form (*symbol-to-rename new-name*).
    - **:self-flavor** — The type of flavor to which this function belongs.

This function is a suitable *place* argument for **setf**; however, casual users should not need to change any of these values directly.

arglist *function* &optional *real-flag-p* Function

This function is given a function object or a function spec and returns its best guess at the nature of the function's lambda list. It can also return a second value that is a list of descriptive names for the values returned by the function. If *function* is a function spec, **arglist** is invoked on its function definition. If the *function* is an actual lambda expression, its cadr, the lambda list, is returned.

Some functions' real argument lists are not what would be most descriptive to a user. A function may take a &rest argument for technical reasons even though there are standard meanings for the first elements of that argument. For such cases, the definition of the function can specify, with a local declaration, a value to be returned when the user asks about the argument list. For example:

```
(defun foo (&rest rest-arg)
 (declare (arglist x y &rest z))
 ...)
```

The *real-flag-p* option allows the caller of **arglist** to indicate that the real argument list should be used even if a declared argument list exists.

By means of a **values** declaration in the function's definition, entirely analogous to the **arglist** declaration above, you can specify a list of mnemonic names for the returned values. This list is then returned by **arglist** as the second value:

```
(arglist 'arglist)
=> (function &optional sys::real-flag)
   (arglist return-list)
```

Because this information is readily available from a Lisp Listener or the Zmacs editor when you press CTRL-SHIFT-A, it is worthwhile to provide good mnemonic names in these declarations.

function-name *function* &optional *try-flavor-name-p* Function

This function returns the function spec that is the name of the function specified by *function*, if that can be determined. The *function* argument can be either a function spec or a function object. If *function* does not describe what its name is, the original *function* argument is returned.

If *try-flavor-name-p* is true and if *function* is a flavor instance (which can, after all, be used as a function), then the flavor name is returned. If the *try-flavor-name-p* argument is **nil**, flavor instances are treated as anonymous and the original *function* argument is returned.

sys:args-desc *function-spec* Function

This function returns four values to describe *function-spec*:

- The minimum number of arguments required for a function call
- The maximum number of arguments this function can be called with
- Whether this function uses &rest arguments
- Whether this function has quoted arguments (is a special form)

For example:

```
(sys:args-desc 'get)     => 2 3 nil nil
(sys:args-desc 'quote)   => 1 1 nil t
(sys:args-desc 'funcall) => 1 1 t nil
```

eh:arg-name *function arg-number* Function

This function returns the name of the argument number specified by *arg-number* in *function*. The *function* argument must be a compiled function definition. The returned value is **nil** if the function does not have such an argument or if the name is not recorded. The &rest arguments are not obtained with *arg-number*; use **eh:rest-arg-name** to obtain the name of the &rest argument of *function*, if any.

eh:rest-arg-name *function* Function

This function returns the name of the &rest argument of *function*, or **nil** if *function* does not have one.

eh:local-name *function local-number* Function

This function returns the name of the local variable number specified by *local-number* in the function definition of *function*. If *local-number* is 0, this function retrieves the name of the &rest argument in any function that accepts a &rest argument. If the function does not have such a local variable, **nil** is returned.

To examine a function's documentation string, see **documentation** in the *Explorer Tools and Utilities* manual.

Encapsulations

16.12 The definition of a function spec actually has two parts: the *basic definition* and *encapsulations*. The basic definition is created by functions such as **defun**, and encapsulations are additions made by **trace** or **advise** to the basic definition. The purpose of making the encapsulation a separate object is to keep track of what was made by **defun** and what was made by **trace**. If **defun** is invoked a second time, the old basic definition is replaced by a new one while leaving the original encapsulations intact.

Only advanced users should ever need to use encapsulations directly via the primitives explained in this paragraph. The most common operations with encapsulations are provided as higher-level, easier-to-use features: **trace**, **breakon**, and **advise**.

The actual definition of the function spec is the outermost encapsulation; this definition contains the next encapsulation, and so on. The innermost encapsulation contains the basic definition. An encapsulation is actually a function whose debug-info property list contains the following property and values:

(**sys:encapsulated-definition** *uninterned-symbol encapsulation-type*)

You can recognize a function as an encapsulation by the presence of such an element in the debug-info property list. An encapsulation is usually an interpreted function (a list starting with **named-lambda**), but it can be a compiled function instead if the application that created it wants to compile it.

The *uninterned-symbol*'s function definition is the object that the encapsulation contains and is usually the basic definition of the function spec. It can also be another encapsulation that has in it another debugging info item containing another uninterned symbol. Within this nesting eventually resides a function that is not an encapsulation; this function does not have the sort of debugging info item that all encapsulations have. This function is the basic definition of the function spec.

Literally speaking, the definition of the function spec is the outermost encapsulation. The basic definition is not the definition of the function spec. If you are asking for the definition of the function spec because you want to apply it, you want the outermost encapsulation. But the basic definition can be found mechanically from the definition by following the debugging info fields. Thus, it makes sense to think of the basic definition as a part of the definition of the function spec. In regard to the function-defining forms such as **defun**, it is convenient to think of the encapsulations as connecting between the function spec and its basic definition.

sys:encapsulate *function-spec outer-function type body-form extra-debugging-info* Macro

All the subforms of this macro are evaluated. In fact, the macro could almost be replaced with an ordinary function, except for the way in which *body-form* is handled.

The *function-spec* argument evaluates to the function spec whose definition the new encapsulation should become. The *outer-function* argument should often be the same as *function-spec*. Its only purpose is to be used in any error messages from **sys:encapsulate**.

The *type* argument evaluates to a symbol that identifies the purpose of the encapsulation and indicates what the application is. For example, the application could be **advise** or **trace**. The list of possible types is defined by the system because encapsulations are supposed to be kept in an order according to their type (see **sys:encapsulation-standard-order**). When the basic function is printed with **pprint**, there is a default routine for printing a representation of an encapsulation. However, you can supply your own print routine for each type of encapsulation by specifying a formatting function as the value of the **sys:encapsulation-pprint-function** property on the property list for the symbol *type*. Read the source code for **pprint** for details.

The *body-form* argument evaluates to the body of the encapsulation definition, that is, the code to be executed when it is called. The backquote syntax is typically used for this expression (see Section 18, Macros). The **sys:encapsulate** form is a macro because, while *body-form* is being evaluated, the **sys:encapsulated-function** variable is bound to a list of the form (**function** *uninterned-symbol*), referring to the uninterned symbol used to hold the prior definition of *function-spec*. If **sys:encapsulate** were a function, *body-form* would simply be evaluated normally by the evaluator before **sys:encapsulate** is invoked; thus, there would be no opportunity to bind **sys:encapsulated-function**. The *body-form* should contain `(**apply ,sys:encapsulated-function arglist**) if the encapsulation is to actually call the original definition, which, of course, may include other encapsulations as well. The variable **arglist** is bound by some of the code that the **sys:encapsulate** macro produces automatically. When the body of the encapsulation is run, **arglist**'s value is the list of the arguments that the encapsulation received.

The *extra-debugging-info* argument evaluates to a list of extra items to put into the debugging info field of the encapsulation function (besides the one starting with **sys:encapsulated-definition**, which every encapsulation must have). Some applications find this list useful for recording information about the encapsulation for their own use later.

If **compile-encapsulations-flag** is not **nil**, the encapsulation is compiled before it is installed. The encapsulations on a particular function spec can be compiled by calling **compile-encapsulations**. Compiled encapsulations can still be unencapsulated because the information needed to do so is stored in the debugging info field, which is preserved by compilation. However, applications that wish to modify the code of the encapsulations they previously created must check for encapsulations that have been compiled and uncompile them. This can be done by finding the **:interpreted-definition** entry in the debugging info field, which is present in all compiled functions except for those made by file-to-file compilation.

When a special form is encapsulated, the encapsulation is itself a special form with the same argument quoting pattern. Therefore, when the outermost encapsulation is started, each argument has been evaluated or not evaluated as appropriate. Because each encapsulation calls the next definition with **apply**, no further evaluation takes place, and the basic definition of the special form also finds the arguments evaluated or not evaluated as appropriate. The basic definition may call **eval** on some of these arguments or on parts of them; the encapsulations should not.

Macros cannot be encapsulated, but their expander functions can be. If the definition of *function-spec* is a macro, then **sys:encapsulate** automatically encapsulates the expander function instead. In this case, the definition of the uninterned symbol is the original macro definition, not just the original expander function. The encapsulation cannot apply the macro definition. Thus, during the evaluation of *body-form*, **sys:encapsulated-function** is bound to the form (**cdr** (**function** *uninterned-symbol*)), which extracts the expander function from the prior definition of the macro.

Because only the expander function is actually encapsulated, the encapsulation does not see the evaluation or execution of the expansion itself. The value returned by the encapsulation is the expansion of the macro call, not the value computed by the expansion.

A program that creates encapsulations often needs to examine an encapsulation it created and find the body. For example, adding a second piece of advice to one function requires this operation. The proper way to perform such an operation is with **sys:encapsulation-body**.

sys:encapsulation-body *encapsulation* Function

This function returns a list whose car is the body form of *encapsulation*. This list is the form that was the fourth argument of **sys:encapsulate** when *encapsulation* was created. To envision this relationship, consider the following:

```
(sys:encapsulate 'foo 'foo 'trace 'body))

(sys:encapsulation-body (fdefinition 'foo))
=> (body)
```

One function can have multiple encapsulations created by different subsystems. In this case, the order of encapsulations is independent of the order in which they were made. The order of the encapsulations depends on their types. All possible encapsulation types have a specified order, and a new encapsulation is put in the proper place among the existing encapsulations according to its type and the types of the existing encapsulations.

sys:encapsulation-standard-order Variable

The value of this variable is a list of the allowed encapsulation types in the order in which the encapsulations are supposed to be kept (innermost encapsulations first). If you want to add new kinds of encapsulations, you should add another symbol to this list, whose initial value is the following:

(advise breakon trace sys:rename-within)

The items in this list are used as follows: **advise** encapsulations are used to hold advice; **breakon** encapsulations are used for implementing **breakon**; **trace** encapsulations are used for implementing tracing; and **sys:rename-within** encapsulations are used to record the fact that function specs of the form (**:within** *within-function altered-function*) have been defined. In this last item, the encapsulation goes on *within-function*.

Every symbol used as an encapsulation type must be on the list **sys:encapsulation-standard-order**. Additionally, it should have a **sys:encapsulation-pprint-function** property whose value is a function that **pprint** calls to process encapsulations of that type. This function need not take care of printing the encapsulated function because **pprint** will do so itself. However, this function should print any information about the encapsulation that the user ought to see. Refer to the code for the printing function for **advise** to see how to write one.

To find the correct place in the ordering for inserting a new encapsulation, you must parse the existing ones. This parsing can be done with the function **sys:unencapsulate-function-spec**.

sys:unencapsulate-function-spec *function-spec* &optional *encapsulations* Function

This function takes one function spec and returns another. If a function has been encapsulated by more than one encapsulation, then you can choose which encapsulations are to remain and which you want eliminated. This choice is provided by the &optional *encapsulations* argument, which should be a list of encapsulation names. This argument informs the **sys:unencapsulate-function-spec** function of which encapsulations are not to be eliminated. If this argument is not provided, then all encapsulations of *function-spec* are eliminated.

Actually, this function takes the argument *function-spec* and returns only the basic function definition and any encapsulations in the argument *encapsulations*. If the original function spec is undefined or has only a basic function definition (that is, its definition is not encapsulated), then the original function spec is returned unchanged. For example:

```
;; Define a function called foo.
 (defun foo (x)
    (+ x 2))

;; Now encapsulate it with trace.
 (trace foo)

;;Note the tracing in its evaluation.
 (foo 20)
 => (1 enter foo: 20)
    (1 exit foo: 22)
    22
;; Now unencapsulate foo.
 (fset 'foo (sys:unencapsulate-function-spec 'foo))
 => foo

;; Foo now does not get traced during evaluation.
 (foo 20)
 => 22
```

Of course, all system-defined encapsulations have specific unencapsulation functions (**trace** has **untrace**) that use **sys:unencapsulate-function-spec**. But if you are using a user-defined encapsulation function that does not currently have an unencapsulation function, you can use **sys:unencapsulate-function-spec** to unencapsulate the function.

sys:rename-within-new-definition-maybe *function-spec new-structure* Function

At some point, you may have a basic function that has several (or no) **sys:rename-within** encapsulations operating on it. The **sys:rename-within-new-definition-maybe** function performs the surgical **rplacd**'s on the basic form so that the proper renamed functions are called. Once these replacements are finished, the modified *new-structure* is stored as the basic function definition. The altered (copied) list structure is returned.

It is not necessary to call this function when you replace the basic definition because **fdefine** with *carefully* specified as true does so for you. The **sys:encapsulate** macro does this to the body of the new encapsulation. Thus, you need only call **sys:rename-within-new-definition-maybe** yourself if you are replacing part of the definition.

For proper results, *function-spec* must be the outer-level function spec. That is, the value returned by **sys:unencapsulate-function-spec** is *not* the right function spec to use. This value will have had one or more encapsulations stripped off, including the **sys:rename-within** encapsulation, if any. Thus, no renamings are performed.

Function Predicates

16.13 The following functions determine if an object is a function, a compiled function, or a special form.

functionp *object* *[c]* Function
functionp *object* &optional *allow-special-forms-p* Function

This function returns true if *object* is a function (essentially, something that is acceptable as the first argument to **apply**); otherwise, it returns **nil**. In addition to interpreted, compiled, and microcoded functions, **functionp** is true of closures and symbols whose function definitions are **functionp**.

The **functionp** function is not true of objects that can be called as functions but are not normally thought of as functions: arrays, stack groups, and flavor instances. As a special case, **functionp** of a symbol whose function definition is an array returns true, because in this case the array is being used as a function rather than as an object.

As an Explorer extension, if *allow-special-forms-p* is specified and true, then **functionp** is true of macros and special forms. Normally, **functionp** returns **nil** for these because they do not behave like functions.

compiled-function-p *object* *[c]* Function

This predicate returns true if *object* is a compiled code object (either a function or a special form). Otherwise, it returns **nil**.

compiledp *function* &optional *dont-unencapsulate* Function

This predicate returns a true value if *function* is a compiled function and **nil** otherwise. The *function* argument should be a function or a function spec. If the original interpreted definition is known, then this is the value returned. If *dont-unencapsulate* is true, then **sys:unencapsulate-function-spec** is not applied to *function*.

fboundp *symbol* *[c]* Function

This predicate returns true if the function definition cell of *symbol* has a definition; this includes macro and special form definitions. To test for macros and special forms but not functions, use the **macro-function** and **special-form-p** predicates.

See also **symbol-function** and **fmakunbound** in paragraph 2.8, Function Definition Cell.

special-form-p *symbol* *[c]* Function

This predicate tests *symbol* to see if it names a special form. If it does, it returns true; otherwise, it returns **nil**.

Note that a *symbol* that returns true for **special-form-p** may also return true for **macro-function** because any macro can be implemented as a special form to speed up processing.

CLOSURES

17

Closure Definitions

17.1 The data type **closure** defines a functional object useful for implementing certain advanced access and control structures. The primary difference between closures and ordinary functions is that closures maintain state information in the form of variable bindings, which are typically thought of as existing outside of the function definition.

For instance, if you wanted to write a function to keep count of the number of times it was called, you could create a closure that would record in a closure variable the number of calls to that function. This closure variable is made available each time the closure function is called and is saved each time the closure is exited. Access to such closure variables is limited to their corresponding closure functions, ensuring integrity of the state information.

The two kinds of closures available on the Explorer system are *lexical* closures (as defined by Common Lisp) and *dynamic* closures (which are provided as an Explorer extension). With lexical closures, the values of all the apparent local variables within the lexical scope of the closure function are saved. With dynamic closures, the values of special variables are saved; you must explicitly list which special variables you want affected.

Note that dynamic closures are a holdover from Zetalisp and are not recommended for use in new programs.

Dynamic Closures

17.1.1 The **closure** function allows you to save a particular set of bindings for a specified set of special variables. The following form, in which *var-list* is a list of special variables and *function* is any function, creates and returns a dynamic closure:

(`setf` *clsr* (`closure` *var-list function*))

When this form is invoked, the values of the variables in *var-list* at the point of invocation are saved, becoming closure values, and the function definition of *function* is used as the closure function definition.

When this closure is applied to arguments, all of the current bindings of the variables in *var-list* are saved away, and the values that were present when the closure was created, or the last time the closure was applied to arguments, are bound to the symbols in *var-list*. Then *function* is applied to the arguments.

Consider the following example:

```
(defvar x 1)                  ; Establish a global special variable and give it a
                              ; value of 1.
(defun foo ()                 ; Define a function foo.
   (setf x (+ x 2)))

(setf close-foo               ; Establish a closure called
 (let ((x 10))                ; close-foo, giving x an initial
   (closure '(x) 'foo)))      ; closure value of 10.

(funcall close-foo) => 12     ; The closure close-foo is applied using the
                              ; closure value of x, 10. The closure value
                              ; of x becomes 12.
(funcall close-foo) => 14     ; The closure value of x becomes 14.

(foo) => 3                    ; Foo uses the global value of x,
                              ; which is 1, and sets it to 3.
```

A dynamic closure can be made around any function spec that evaluates to a function object. The form could evaluate to a function, as with `(function (lambda () x))`. In the example above, the function spec is `'foo`, and it evaluates to the symbol `foo`. It is usually better to use a symbol that is the name of the desired function so that the closure points to the symbol. Then, if the symbol is redefined, the closure uses the new definition. If you actually prefer that the closure continue to use the old definition that was current when the closure was made, use the #' Reader macro, as in the following:

```
(closure '(x) #'foo)
```

Explicit closures made with **closure** record only the dynamic bindings of the specified variables. Another closure mechanism is activated automatically to record lexical bindings whenever **function** is used around an explicit lambda expression, but **closure** itself has no interaction with lexical bindings.

It is the user's responsibility to ensure that the bindings the closure is intended to record are dynamic bindings, either by means of **special** declarations (see paragraph 13.3, Declaration Specifiers) or by making the variables globally special, as shown in the previous example with **defvar**. If the function closed over is an explicit lambda expression, it is occasionally necessary to use declarations within it to ensure that the variables are considered dynamic there. But this is not needed if the variables are globally special or if a special declaration is visible where **closure** is called.

Usually, the compiler can tell when a special declaration is missing, but when making a closure, the compiler detects this absence only after acting on the assumption that the variable is lexical. Then it is too late to fix the problem. The compiler warns you if this happens.

With dynamic closures, lambda binding never really allocates any storage to create new bindings. Bindings receive separate storage only when the **closure** function itself finds they need it. Thus, there is no cost associated with closures when they are not in use.

Closure implementation involves two kinds of value cells. Every symbol has an *internal value cell*, which is where its dynamic value is normally stored. When a variable is closed over by a closure, the variable receives an *external value cell* to hold its value. The value in the external value cell is found through the usual access mechanisms (such as evaluating the symbol, calling **symbol-value**, and so on) because the internal value cell is made to contain a forwarding pointer to the external value cell that is current. Such a forwarding pointer is present in a symbol's value cell whenever its current binding is being remembered by a closure; at other times, there is no invisible pointer, and the value resides directly in the symbol's internal value cell.

Lexical Closures

17.1.2 A lexical closure in Common Lisp is returned by the **function** special form when a lambda expression is passed to it as an argument. The lambda expression can reference lexically bound variables outside the lexical scope of the lambda expression. A lexical closure saves the entire lexical environment of the lambda expression for future processing. For example:

```
(defun foo (x y z)                  ; Define foo.
  (function (lambda (a b)           ; Return a lambda expression.
              (+ a b x y z))))
(setf bar (foo 1 2 3))              ; Bar becomes a lexical closure and the
                                    ; variables x, y, and z are saved after the
                                    ; execution of (foo 1 2 3).
(funcall bar 0 0) => 6              ; Bar executes with variables x, y, and z
                                    ; still bound to 1 2 3.
(funcall bar 1 1) => 8
(funcall bar 4 5) => 15
```

In the `setf` form, `bar` is set to a lexical closure resulting from the invocation `(foo 1 2 3)`. Therefore, the lexical variables of `foo`, `x`, `y`, and `z` remain bound to `1`, `2`, and `3`, even after `foo` completes executing. When `bar` is invoked in the `funcall` forms, the variables `x`, `y`, and `z` have the values they assumed when the closure was created, or the values they had the last time the called closure was exited. The parameters `a` and `b` are bound within this environment to the arguments; the function then executes. For example:

```
(defun foo (x y)                    ; Define foo.
  (function (lambda ()              ; Return a lambda expression.
             (list x y (setf y (+ y 1))))))

(setf bar (foo 1 2))                ; Bar becomes a lexical closure with x bound
                                    ; to 1 and y bound to 2.
(funcall bar) => (1 2 3)            ; Variable y is incremented by 1 and retains
(funcall bar) => (1 3 4)            ; this binding.
(funcall bar) => (1 4 5)
```

In this example, `bar` becomes a lexical closure over the variables `x` and `y`. Each time `bar` is invoked, it changes the variable `y` to an incremented value.

When more than one closure is created within the same lexical environment and they share the same closure variables, then any resetting of shared closure variables pervasively affects all closures within the environment. For example:

```
(defun two-lambdas (a)                  ; Define two-lambdas.
  (list (function (lambda () a))        ; Return a list of two closures.
      (function (lambda (b)
                   (setf a b)))))

 (setf funcs (two-lambdas 3))           ; Bind variable a to 3 and set funcs to
                                        ; the list of two closures.
 (setf fun1 (first funcs))              ; Set fun1 to the first closure.
 (setf fun2 (second funcs))             ; Set fun2 to the second closure.
 (funcall fun1) => 3                    ; fun1 returns the value of a.
 (funcall fun2 20) => 20                ; fun2 changes a and returns the value.
 (funcall fun1) => 20                   ; Variable a is also changed in fun1.
```

In this example, two closures are created and returned by the function `two-lambdas`. When the second closure is evaluated within `fun2`, it resets the closure lexical variable `a`, and this resetting also affects the value of `a` in the closure `fun1`. This effect is produced because the two different closures are created within one environment; therefore, only one closure variable is created and both closures use the same closure variable.

Functions That Manipulate Dynamic Closures

17.2 For dynamic closures, the use of the following functions is fairly straightforward. For lexical closures, these routines tend to reveal details of the implementation and are of little use, except for **closure-function** and **closeurep**.

closure *var-list function* Function

This function creates and returns a closure of *function* over the variables in *var-list*. Note that all variables on *var-list* must be special if the function is to compile correctly. The *function* argument can be a function object or a function spec.

symeval-in-closure *closure symbol* Function

This function returns the binding of *symbol* in the environment of *closure*; that is, it returns what you would get if you restored the bindings known by *closure* and then evaluated *symbol*. This function allows you to *look around inside* a closure. If *symbol* is not closed over by *closure*, this function is exactly like **symbol-value**.

set-in-closure *closure symbol new-value* Function

This function sets the binding of *symbol* in the environment of *closure* to *new-value*; that is, it does what would happen if you restored the bindings known by *closure* and then set *symbol* to *new-value*. This function allows you to change the contents of the bindings known by a closure. If *symbol* is not closed over by *closure*, this function is exactly like **set**.

locate-in-closure *closure symbol* Function

This function returns the location of the place in *closure* where the saved value of *symbol* is stored. The following is an equivalent form:

(locf (symeval-in-closure *closure symbol***))**

boundp-in-closure *closure symbol* Function

This function returns true if the binding of *symbol* in *closure* is not void. This value is what (**boundp** *symbol*) would return if executed in the saved environment of *closure*.

makunbound-in-closure *closure symbol* Function

This function causes the binding of *symbol* in *closure* to be void. This is what (**makunbound** *symbol*) would do if executed in the saved environment of *closure*.

closure-alist *closure* Function

This function returns an association list of (*symbol* . *value*) pairs describing the bindings that the closure performs when it is called. This list is not the same one that is actually stored in the closure; that list contains pointers to value cells rather than symbols, and **closure-alist** translates them back to symbols so that you can understand them. As a result, changing this list does not change the closure.

The list that is returned can contain unbound cells if some of the closed-over variables are unbound in the closure's environment. In this case, printing the value produces an error (accessing a cell that contains an unbound marker is always an error unless performed in a special, careful way), but the value can still be passed around.

closure-variables *closure* Function

This function returns a list of variables closed over in *closure*. This list is **equal** to the first argument specified to the function **closure** when this closure was created.

closure-function *closure* Function

This function returns the closed function from *closure*. This closed function is the function that was the second argument to **closure** when the closure was created. If a symbol was passed to *closure*, a symbol is returned. If a function object was passed to *closure*, that object is returned.

closure-bindings *closure* Function

This function returns the actual list of dynamic bindings to be performed when *closure* is entered.

copy-closure *closure* Function

This function returns a new closure that has the same function and variable values as *closure*. The bindings are not shared between the old closure and the new one so that if the old closure changes a closed variable's value, the value in the new closure does not change.

let-closed ({*var*|(*var value*)}*) *function* Macro

When you are using dynamic closures, it is very common to bind a set of variables with initial values only to make a dynamic closure over those variables. Furthermore, the variables must be declared special. The **let-closed** macro does all of this. It is best described by example:

```
(let-closed ((a 5) b (c 'x))
  (function (lambda () ...)))
; macro-expands into
(let ((a 5) b (c 'x))
  (declare (special a b c))
  (closure '(a b c)
    (function (lambda () ...))))
```

Note that the following code, which would often be useful, does not work as intended if `x` is not special outside the `let-closed`:

```
(let-closed ((x x))
 (function ...))
```

This code does not behave as expected because the reference to `x` as an initialization for the new binding of `x` is affected by the special declaration that the `let-closed` produces. It does not see any lexical binding of `x`. This behavior is unfortunate, but it is required by the Common Lisp specifications. To avoid the problem, write the following:

```
(let ((y x))
 (let-closed ((x y))
   (function ...)))
```

You can also avoid the problem by simply changing the name of the variable outside the `let-closed` to something other than `x`.

closurep *object* Function

This predicate returns true if its argument is a closure; otherwise, it returns **nil**.

18

MACROS

Macro Definitions

18.1 A *macro* is a function written by the programmer to return a Lisp form that is subsequently evaluated. In contrast to functions, macros involve an extra level of algorithm abstraction. When a function is called, it performs some side effects and returns a value. When a macro is called, an expansion function, which should have no side effects, generates another piece of code (the expansion form), which is subsequently evaluated. The expansion form can produce side effects, and its returned value is that value returned from the macro call.

When you define a macro, you are actually writing the expansion function. When defining a macro, keep in mind the following issues:

- The returned value of the expansion function should be a Lisp form that actually performs the implied work of the macro call.
- The arguments received by the expansion function are the unevaluated arguments in the macro call.
- The expansion function does *not* execute in the local environment of the macro call.
- The generated expansion form *does* execute in the local environment of the macro call.
- When macro calls are compiled, the expansion function is called at compile time, but only the expansion form becomes part of the compiled object.

Each of these issues is discussed in greater detail later in this section.

Advantages of Macros

18.1.1 Because of the extra layer of abstraction involved with macros, the macro writer can make the expansion function more general in purpose without making the expansion form less efficient at run time. For instance, the **setf** macro is one of the most general forms in Common Lisp, yet the expansion form that it generates is very specific based on the kind of *place* argument it receives.

Sometimes macros are written to hide complexity. For instance, assume that you use a complex form frequently, but for your purposes, some of the arguments are invariant. You could write an intermediate function to provide a more efficient interface to the more complex form, but you would be adding the expense of an extra function call at run time. A macro, on the other hand, would not generate the extra function call.

Macros are frequently used to create **def*name*** forms that are meant to execute at the Lisp top level. If the purpose of the **def*name*** form is to perform some bookkeeping and to use one or more other **def*other-name*** forms, the macro expansion form can bundle these forms together in a **progn** that executes each form as if it were at top level (assuming that the **def*name*** form is executed at top level).

Macro Expansion

18.1.2 Macro calls and function calls are actually quite different, despite their similarity in appearance. You cannot apply macros to arguments by using functions such as **funcall** or **apply**; you can only **eval** macros. In practice, when the Lisp evaluator encounters a cons whose car is a symbol, it first checks to see if the symbol is defined locally (via **flet**, **labels**, or **macrolet**). If not, it searches for a global definition. If the global definition is a cons whose car is the symbol **macro**, then the cons is a macro definition. At this point, the evaluator knows that the original form was a macro call.

The cdr of the macro definition is defined to be an expansion function that has two parameters: the first argument should be the original macro call form; the second argument should be a lexical environment. The value returned from the expansion function is another Lisp form that is subsequently evaluated, thus performing the desired action.

When the expansion function is called, it receives the arguments unevaluated, which is roughly equivalent to using &quote in special forms. However, special forms may eventually have to explicitly call **eval** to have an argument evaluated. Macros, in contrast, need only construct an expansion that could implicitly evaluate an argument.

The compiler must know about a macro definition before it sees the reference to it; otherwise, it assumes that the macro call is a function call. For the interpreter, the macro is required to exist only at the time of the expansion call.

As an optimization, the macro expansion can be remembered in interpreted mode such that the expansion function need not be called the next time that this particular macro call is evaluated. Thus, the overhead of the macro expansion is handled only once at run time for each macro call (see paragraph 18.5, Displacing Macro Calls). In compiled mode, however, the macro expansion is simply coded inline at compile time, thus completely avoiding the expansion overhead at run time. Both of these optimizations have the drawback that if the macro changes, the previously existing expansions may be incorrect. Consequently, interpreted functions must be **redefuned**, and compiled functions must be recompiled with the new macro definition.

Because macro expansions are compiled inline, they can pose a special problem. Be aware that macros must be changed in an upwardly compatible manner, unless you intend to recompile all source files that use the macro. When you load object files that were expanded with old versions of macros, the loader prints a message to that effect. The best approach to avoid this problem is to collect macros into a single file and make sure that they are precompiled and preloaded before you compile any source files that reference them. See Section 23, Maintaining Large Systems.

Defining Macros

18.2 The following forms are used for creating user-defined macros and for determining if a particular function spec is already defined as a macro.

macro *name (form-arg env-arg) {exp-form}** Special Form

This is the Explorer primitive special form for defining macros. The *name* can be any function spec. The *form-arg* and *env-arg* arguments must be variables. The *form-arg* variable is set to the original calling form. The *env-arg* variable is set to the local environment structure. The *exp-forms* argument is a sequence of Lisp forms that constitutes the expansion function of the macro; the last form should return the expanded code.

Only very sophisticated macros need to use the *env-arg* parameter, and the information on it can be found in the description of **defmacro** under the discussion of the &environment lambda-list keyword. If you are not going to use this parameter, then declare the variable name that you use to be ignorable; for example, `(declare (ignore env-arg))`.

defmacro *name lambda-list {declaration | doc-string}* {exp-forms}** *[c]* Macro

This macro is a general-purpose macro-defining macro. The *name* argument is the name of the macro to be defined and is the returned value. It can be any function spec, but normally it is not useful to define anything except a symbol, because the evaluator looks only to symbol definition cells for macro definitions. However, sometimes it is useful to define a **:property** function spec as a macro when some part of the system (such as **locf**) looks for an expansion function on a property.

When a macro defined by **defmacro** is called, *lambda-list* is matched against the cdr of the call form; the elements of *lambda-list* are bound to *unevaluated* arguments of the call form. The bound variables in *lambda-list* are to be used as variables within the body of the expansion function *exp-forms*. The *exp-forms* of the expansion function are evaluated with these bindings in effect, and the result is returned to the evaluator as the expanded code of the macro.

If you include a *doc-string* argument, it must be followed by a declaration or at least one *exp-form*. Otherwise, it is treated as an *exp-form*. The documentation string is associated with the function spec of *name* with the documentation type of **function**. This documentation can be accessed with the following form:

```
(documentation name 'function)
```

It can be updated with this form:

```
(setf (documentation name 'function) "new doc-string")
```

The *lambda-list* is like a lambda list to a function but has some significant differences. Note the following macro definition example and its invocation form (disregard for now the part of the macro definition that computes the expansion function; it is explained later):

```
;;; Define the macro called "for" that is similar to the iteration instruction in BASIC.
(defmacro for (var low high &body remaining-forms)
  ...)
;;; Now call "for" that prints all integers (and their squares) between 1 and 100.
(for a 1 100
     (print a)
     (print (* a a)))
```

The parameter binding in *lambda-list* is as follows: `var` is bound to `a`; `low` is bound to `1`; `high` is bound to `100`; and `remaining-forms` is bound to the list `((print a) (print (* a a)))`.

Lambda lists in macro definitions have a destructuring feature not available in function definitions. If the macro writer knows that a particular argument being passed will always be a list, then the corresponding parameter position in the lambda list can be a nested lambda list. The nested lambda list matches elements from the argument list to parameters specified in the nested lambda list. For example:

```
(defmacro person-id ((first-name last-name) age occupation)
  ...)

(person-id (john doe) 32 'programmer)
```

In this example, `first-name` is bound to `john` and `last-name` is bound to `doe`. No extra restrictions are placed on the nested lambda lists. Because destructuring can be used to create mappings to potentially complex tree structure arguments, macro lambda lists allow a lambda list to be a dotted list, in which the last symbol after the dot is bound to the remaining arguments at that level. This operation is equivalent to using &rest except that it does not allow subsequent lambda-list keywords, such as &aux. Visually, a dotted list may be easier to follow when destructuring is used.

Another major difference between a lambda list of a macro and a lambda list of a function is the use of lambda-list keywords. Some of the keywords in a function lambda list cannot be used in a macro lambda list, and some of the keywords in a macro lambda list cannot be used in a function lambda list.

The only lambda-list keywords that can be used in both functions and macros are &optional, &rest, &key, &allow-other-keys, and &aux, which are all standard Common Lisp. They operate the same way for macros as they do for functions and are described in paragraph 16.1.2, Lambda-List Keywords.

The following are lambda-list keywords that can be used only in macro definitions:

&body *[c]* Lambda-List Keyword

This keyword is identical to the &rest keyword, except that it tells certain pretty-printing and editing functions to treat the remaining subforms of the calling form as a *body* rather than as *arguments* and to indent them accordingly. Note that the list supplied with the &rest or &body lambda-list keyword is allowed to be the original list of the caller, so the macro expansion function should not alter the list.

The &body lambda-list keyword also has the following syntax:

&body (*body-var* [*declarations-var* [*doc-string-var*]])

If *declarations-var* and *doc-string-var* are not supplied, then this syntax is the same as the normal syntax without any parentheses around *body-var*. Otherwise, the *body-forms* supplied in the macro call are passed to **parse-body** and the return values are set to *body-var*, *declarations-var*, and *doc-string-var*, respectively. This procedure is merely a convenience to macro writers so that they will not have to bother with getting an environment value and calling **parse-body** themselves.

&whole *[c]* Lambda-List Keyword

This keyword causes the variable that follows it to be bound to the entire macro call form. This lambda-list keyword can be used in addition to other lambda-list keywords but should come first.

&environment *[c]* Lambda-List Keyword

This keyword causes the variable that follows it to be bound to the *local macros environment* of the macro call being expanded. This binding is useful if the code for expanding this macro needs to invoke **macroexpand** on subforms of the macro call. Then, to achieve correct interaction with **macrolet**, the local macro environment should be passed to **macroexpand** as its second argument. Thus, you would write the following:

```
(defmacro name (parms &environment env)
  . . .
  (macroexpand form env)
  . . .)
```

In this form the environment is received in the *env* parameter. The environment value of this parameter contains the definitions of any macros defined by **macrolet**. The macro writer does not need to know the structure of an environment argument.

&list-of — Lambda-List Keyword

This keyword is an Explorer extension. This keyword provides another way of destructuring macro-calling arguments.

The object following the &list-of keyword should be a list whose elements are bound to a list of corresponding elements in a call form. This requirement is best explained by example. Note the following macro definition and call form (again, disregard the part of the macro that computes the expansion function):

```
;;; Define a macro that sends commands to a turtle's brain, which is
;;; represented by an array.
(defmacro send-commands (send-by &body
          &list-of (command . arguments))
     ...)
;;; Now send a set of commands.
(send-commands (aref turtle-mind i)
 (forward 100)
 (beep)
 (left 90)
 (pen 'down 'red)
 (forward 50)
 (pen 'up))
```

In this example, the parameter `send-by` is bound to the list `(aref turtle-mind i)`. Notice that the object following the &list-of keyword in the `defmacro` is the cons `(command . arguments)`. Therefore, the car of this cons, `command`, is bound to a list that contains all the cars of the command lists in the call form. Thus, the value of `command` is the following list:

```
(forward beep left pen forward pen)
```

The cdr of the parameter cons, `arguments`, is bound to the cdrs of all the lists in the remaining call form. Therefore, `arguments` is bound to the following list:

```
((100) nil (90) ('down 'red) (50) ('up))
```

Note that the pattern of `&list-of`'s parameter determines how the bindings occur. The pattern of `&list-of`'s parameter is matched against each of the remaining lists of the call form, and corresponding elements are bound accordingly. In this example, the pattern of `&list-of`'s parameter causes `arguments` to be bound to a list of lists and `commands` to be bound to a list of symbols. This binding occurs because `(commands . arguments)` is a cons, and the car and cdr of this cons constitute the pattern matched with the cars and cdrs of the remaining lists in the call form. If there is no corresponding matching element in any of the remaining lists, then **nil** is assigned as the element for these lists (note that this is the case for the list `(beep)` in the calling form).

If the parameter of `&list-of` is the list `(commands arguments)`, then the pattern matching is element to element instead of car to car and cdr to cdr; therefore, `commands` is bound as it was in the previous example, and `arguments` is bound to the following:

```
(100 nil 90 'down 50 'up)
```

Note that this example uses `&body` in conjunction with `&list-of`; if `&list-of` is used by itself, then its argument in the calling form must be a list of lists.

You can combine the &optional and &list-of keywords for some interesting effects because all of the features of optional parameters, such as *init-forms* and *supplied-p* parameters, can be incorporated into &list-of parameters. Consider the following macro with one optional parameter, which, if supplied, is assumed to be an association list:

```
(defmacro alist-destructure
            (&optional
             &list-of ((keys data)                ;; the parameter name(s)
                       ´((a 1) (b 2) (c 3))       ;; the init-form
                       supplied-p))               ;; supplied-p parameter

  (format nil "keys=~a  data=~a  supplied=~a"
               keys data supplied-p)
)
```

In this example, the purpose of the `format` statement is to clarify the bindings of the various parameters rather than to demonstrate a serious macro expansion function. Now consider the consequences of calling this macro:

```
(alist-destructure) ==> "keys=(a b c) data=(1 2 3) supplied=NIL"

(alist-destructure ((x 100) (y 200) (z 300)))
==> "keys=(x y z) data=(100 200 300) supplied=T"
```

Note, in the preceding example, that when no argument is supplied, the *init-form* `((a 1) (b 2) (c 3))` is used and `supplied-p` is set to `NIL`. The `&list-of` bindings of `key` and `data`, however, operate as if the caller had supplied an argument (just like normal *init-forms*). When an association list argument is supplied, the *init-form* is ignored and the `supplied-p` variable is set to true.

macro-function *function-spec* *[c]* Function

If *function-spec* is defined as a macro, then this function returns its expander function. Otherwise, **macro-function** returns **nil**. Common Lisp specifies that *function-spec* must be a symbol.

Since a definition of a macro on the Explorer system is really a cons of the form (**macro** . *expander-function*), you can retrieve the expander function using (**cdr** (**fdefinition** *function-spec*)), but it is more efficient to use **macro-function**. On the Explorer system, some Common Lisp macros are actually implemented as special forms, such as **cond**. When these symbols defined as special forms are passed as arguments to **macro-function**, the value returned is an *alternate macro definition* expansion function. Thus, portable Common Lisp programs can use these expansion functions; however, compiled functions must use the special forms. The following form is permitted:

(**setf** (**macro-function** *function-spec*) *expander*)

The following is equivalent to the preceding form:

(**fdefine** *function-spec* (**cons** '**macro** *expander*))

Constructing a Macro Expansion

18.3 Recall that the value of a macro expansion is something that is, in turn, evaluated. If the expansion is to return a symbol or some object other than a list, writing the expansion can be quite simple.

Simple Macro Expansion Functions

18.3.1 Simple Lisp objects evaluate to themselves. For example:

```
(defmacro my-name ()
   (string-capitalize-words user-id))    ; Assume the user logs in with
                                         ; his or her last name.
(format nil "My name is ~A. " (my-name))
=> "My name is Smith. "
```

If the expansion is to return a function call form, you must be more creative. Of the two general approaches to this situation, the first is to use the **list** function and have each of its arguments represent the elements in a function call. For example:

```
(defmacro state-my-name ()
   (list 'format 'nil "My name is ~A. " (my-name)))

(state-my-name) => "My name is Smith. "
```

Macro Expansion Using the Backquote

18.3.2 Although the previous approach to constructing macros works properly, the numerous calls to **list** and the need to quote many of the arguments makes this approach rather tedious. The preferred way to write a macro expansion function is by using the backquote (‘) syntax. When a backquote appears at the front of a list or a vector, the subsequent elements of that structure are not evaluated unless they have the following syntax:

- *,expression* — Substitutes the expression value for this element in the backquoted sequence.
- *,@expression* — Same as *,expression* except that if the value returned is a sequence, each element of the sequence is appended into the backquoted sequence.
- *,.expression* — Same as *,@expression* except that the sequence returned from the expression may be destructively modified; **nconc** is used instead of **append**.

Consider the following examples:

```
(setf a 1)
(setf b 2)
(setf c 3)
(setf d '(h i))

'(a b c d) => (a b c d)                 ; simple quoted list
`(a b c d) => (a b c d)                 ; backquoted list with no substitutions
`#(a b c d) => #(a b c d)               ; backquoted vector with no substitutions
`(,a b ,c ,d) => (1 b 3 (h i))          ; backquoted list with substitutions
`#(,a b ,c ,d) => #(1 b 3 (h i))        ; backquoted vector with substitutions
`(,a b ,c ,@d) => (1 b 3 h i)           ; substitution with appending
`#(,a b ,c ,@d) => #(1 b 3 h i)         ; substitution with appending, for a vector
`(a (,a ,b ,c) b ,c ,@d) => (a (1 2 3) b 3 h i)
```

An alternate way of expressing the *,@expression* syntax is by using a dotted list syntax and the comma. For example:

```
`(,a b c ,@d) <==> `(,a b c . ,d)
```

Note that the dot does not mean anything different within the backquoted list. As always, it merely indicates that the cdr of this cons (the one whose car is the symbol `c`) points to the next object, which in this case is another cons cell. Of course, as always, the item after the dot must be the last item specified in the list.

Finally, with the exception of the comma dot, the backquote utility does not modify the arguments it is supplied. The elements in the backquoted list may be **eq** to the elements in the backquote result. Thus, if you modify the backquote result, this result may modify the original arguments (there is no guarantee one way or the other).

The following is a simple example of **defmacro** using the backquote substitution scheme. This macro increments the value supplied by `form`. An optional argument specifies the value to increment `form` by; the default is `1`. The syntax line would be as follows:

increment *form* &optional *delta* Macro

The macro definition would be as follows:

```
(defmacro increment (form &optional (delta 1))
   (if (eql 1 delta)
       `(1+ ,form)
       `(+ ,delta ,form)))
```

This macro attempts some optimizing by calling either the function `1+` or the function `+`, depending on the `delta` argument.

The following example combines several features of macros discussed previously. Assume that you want to write a macro called **defprogrammer**, which is used from the top level. This macro is to keep track of your company's programmers and each programmer's languages. The syntax for this macro is as follows:

defprogrammer *last-name* (*first-name* {*other-name*}*) {*language*}*

This macro keeps a list of the programmers in the ***programmers-list*** variable. It also defines a function called *programmers-name*-**knows-p** that accepts a programming language name as an argument and returns true if the programmer knows that language.

```
(defvar *programmers-list* nil
   "List of programmers.")

(defmacro defprogrammer (last-name (first-name &rest other-names) . languages)
   (let ((symbol-name (intern (string-append last-name "-" first-name "-KNOWS-P"))))

      `(progn
          (defun ,symbol-name (language)
              (declare (sys:function-parent ,last-name defprogrammer))
                 ;; Assume that any self-respecting programmer knows Lisp.
         (member language `(lisp ,@languages) :test #'eq))
          (setf (get ',symbol-name :other-names) ',other-names)
          (push ',symbol-name *programmers-list*)
          )))
```

Note that none of the arguments are **evaled** before being processed by the macro expansion function. The `first-name` and any middle names or aliases are parsed as destructured arguments, making it easier to manipulate their values. Placing the dot before the `languages` parameter is equivalent to `&rest languages`.

The `let` statement performs some pre-processing while the `progn` starts the beginning of what will eventually be the macro expansion. One advantage of using a **progn** in the expanded form (rather than using a **let**, for instance) is that if a **progn** is encountered at the top level, then each form within the **progn** is processed as if it were at the top level. Note that the expanded form must be the returned value of the expansion function.

The `declare` statement is included in the `defun` so that if anyone performs a META-. operation on the function being defined, the appropriate `defprogrammer` form will be accessed. This information is available by calling **sys:function-parent** (see Section 16, Functions, for details).

There are several substitutions performed within the backquoted `progn` using the `,` and `,@` delimiters. Note that for a single backquoted form (the `progn` in this case) all nested forms are checked for substitution possibilities.

Consider the results of the following macro call:

```
(defprogrammer Smith (John Q. "Sonny") pascal fortran)
```

This macro call expands to the following:

```
(progn (defun smith-john-knows-p (language)
          (declare (system:function-parent smith defprogrammer))
          (member language (quote (lisp pascal fortran)) :test #'eq))
       (setf (get (quote smith-john-knows-p) :other-names) (quote (q. "Sonny")))
       (push (quote smith-john-knows-p) *programmers-list*))
```

Multiple and Out-of-Order Evaluation

18.3.3 In general, when you define a new macro that contains one or more argument forms, you must be careful that the expansion evaluates the argument forms the proper number of times and in the proper order. There is nothing fundamentally wrong with multiple or out-of-order evaluation if that is what you really want and if it is what you document your form to do. But if it happens unexpectedly, it can make invocations fail to work as they apparently should.

To avoid multiple evaluation, you can use the **once-only** macro, which is best explained by example:

```
(defmacro test (reference form)
   (once-only (form)
      `(setf ,reference (cons ,form ,form))))
```

This form defines `test` in such a way that the `form` is evaluated only once, and references to `form` inside the macro body refer to that value. The `once-only` macro automatically introduces a lambda binding of a generated symbol to hold the value of the form. Actually, it is even more clever: it avoids introducing the lambda binding for forms whose evaluation is trivial and may be repeated without harm or cost, such as numbers, symbols, and quoted structure, which helps produce more efficient code.

However, the **once-only** macro does not completely or automatically solve the problems of multiple and out-of-order evaluation. It is merely a tool that can solve some of the problems some of the time.

Although the following describes how **once-only** operates, it is much easier to use this macro by imitating the preceding example rather than trying to understand **once-only**'s tricky definition.

once-only ({*var*}*) {*body-form*}* Macro

For this macro, the *vars* are variables, and the *body-form* is a Lisp program that presumably uses those variables. When the form resulting from the expansion of the **once-only** is evaluated, it first inspects the values of each of the *vars*; these values are assumed to be Lisp forms. Each of the variables is then bound either to its current value (if the current value is a trivial form) or to a generated symbol. Next, **once-only** evaluates the *body-forms* in this new binding environment and, when they have been evaluated, it undoes the bindings. The result of the evaluation of the last *body-form* is presumed to be a Lisp form, typically the expansion of a macro. If all of the variables have been bound to trivial forms, then **once-only** simply returns that result. Otherwise, **once-only** returns the result wrapped in a lambda combination that binds the generated symbols to the result of evaluating the respective nontrivial forms.

The effect is that the program produced by evaluating the **once-only** form is coded in such a way that each of the forms that is a value of one of the variables in *var* is evaluated only once, unless the form has no side effects. At the same time, no unnecessary temporary variables appear in the generated code, but the body of the **once-only** is not cluttered with extraneous code to decide whether temporary variables are needed.

Expansion Functions With Consing Side Effects

18.3.4 Normally, macro expander functions do not have any side effects, and the only consing they do is to produce the macro expansion that is returned as the function result. Consequently, when the compiler calls a macro expander, the **default-cons-area** is bound to the compiler's temporary area, which is deallocated at the end of the compilation. If, however, the expander does produce side effects that involve allocating memory that will be referenced after the compilation is finished, then the expander needs to bind **default-cons-area** to a nontemporary area around the code that produces the side effect. For example:

```
(let ((default-cons-area sys:background-cons-area))
   (push info *global-list*))
```

Local Macro Definitions

18.4 The **defmacro** macro is used for defining macros whose names have global scope. You can also create local macro definitions that are in effect within a lexical scope.

macrolet ({(*macro-name lambda-list* {*declaration* | *doc-string*}* {*macro-body-form*}*)}*) {*declaration*}* {*body-form*}* *[c]* Special Form

This special form allows you to define macros that are local to a given function, much the same as **let** allows you to define local variables or **flet** allows you to define local functions. The *macro-names* become local macro definitions that are defined within the lexical scope of the *body-forms*. The *lambda-list*, *declarations*, and *macro-body-forms* have the same meaning as they do for **defmacro**. The **macrolet** special form can be used to lexically shadow global macro or function definitions or simply as a convenience to the programmer.

Keep in mind that the expansion functions for these local macros are executed in the global environment. Specifically, this means that the expansion functions may not access any local definitions, such as parameters, variables created by &aux, **let**, **flet**, and so on. The **macrolet** form can be confusing because at first glance the *macro-body-forms* may seem to be within the lexical scope of enclosing forms and may seem to contain the names of these local definitions. However, you must remember what is being evaluated in the expansion function. For instance, a local variable can appear in a backquoted form as long as it is not prefixed with a comma to force an evaluation. The forms created by the macro expansion execute within the local environment, and thus the expansion may contain references to local definitions. Consider the following example:

```
(defun find-ERA (earned-runs innings-pitched)
   (macrolet ((era-macro (er ip)
                 `(/ (* 9 ,er) ,ip)))
      (format nil "ERA=-3,2f"
                (era-macro earned-runs innings-pitched))
      ))
```

When compiled, the `format` form becomes the following:

```
(format nil "ERA=-3,2f"
            (/ (* 9 earned-runs) innings-pitched))
```

Note the various scoping implications of this example: `era-macro` is only defined within the lexical scope of the `macrolet` body forms (in this case, the only form is the `format` function). The variable bindings `er` and `ip` are parameters to the macro and are defined only within the lexical scope of the macro body forms (in this case, the backquoted expansion function). The `era-macro` is not allowed to directly reference the local variables `earned-runs` and `innings-pitched`; any direct reference to these variables in `era-macro` would have implied a reference to global variables of the same names.

Displacing Macro Calls

18.5 Every time the evaluator sees a macro form, it must call the macro expander to expand the form, which is time-consuming. To speed up processing, the expansion of the macro is recorded automatically by destructively modifying the calling form to logically make the expansion appear inline. If the same form is evaluated again, it can be processed immediately. This procedure uses the function **displace**.

A consequence of the evaluator's policy of displacing macro calls is that if you change the definition of a macro, the new definition does not take effect in any form that has already been expanded. An existing form that calls the macro uses the new definition only if the form has never been evaluated. You can redefine a function (with **defun**) to get rid of the expansion.

Also note that when you use **pprint** to view the interpreted function definition, you will see the original macro call form.

displace *form expansion* Function

This function replaces the car and cdr of *form* so that it looks like the following:

(**sys:displaced** *old-form-copy expansion*)

The argument *form* must be a list. When a form whose car is **sys:displaced** is evaluated, the evaluator simply extracts the *expansion* and evaluates it. The *old-form-copy* argument is a newly consed pair whose car and cdr are the same as the original car and cdr of the *form*; thus, it records the macro call that was expanded.

The **displace** function returns *expansion*.

sys:inhibit-displacing-flag Variable

This special variable, which is normally **nil**, can be set to true to prevent the evaluator from displacing macro calls.

Functions to Expand Macros

18.6 The following functions and variable are provided to allow you to control expansion of macros; they are often useful for the writer of advanced macro systems and in tools to examine and understand code that may contain macros.

macroexpand-1 *form* &optional *local-macros-environment* *[c]* Function

If *form* is a macro form, this function expands it (once) and returns the expanded form. Otherwise, it merely returns *form*. The second value is **t** if *form* has been expanded.

The *local-macros-environment* argument is a data structure that specifies the local macro definitions (made by **macrolet**) to be used for this expansion, in addition to the global macro definitions (made by **defmacro** and recorded in function cells of symbols). When **macroexpand-1** is called by the evaluator, this argument comes from the evaluator's own data structures set up by any **macrolet** forms that *form* was found within. When **macroexpand-1** is called by the compiler, this argument comes from data structures kept by the compiler in its handling of **macrolet**.

Sometimes macro definitions call **macroexpand-1**; in this case, if *form* is a subform of the macro call, an &environment argument in the macro definition can be used to obtain a value to pass as *local-macros-environment* (see **macrolet**, described in paragraph 18.4, Local Macro Definitions).

If *local-macros-environment* is omitted or **nil**, only global macro definitions are used.

macroexpand *form* &optional *local-macros-environment* *[c]* Function

If *form* is a macro form, this function expands it repeatedly until the car of the expansion is no longer a macro form and returns the expansion. Otherwise, it returns *form*. The second value is **t** if one or more expansions have taken place.

The *local-macros-environment* argument operates the same way as for **macroexpand-1** (described previously).

In the following example, assume that `setf` is a macro that expands to a **setq** form:

```
(defmacro setf2 (varl var2 data)
     `(setf ,varl (setf ,var2 ,data)))
(macroexpand '(setf2 x y 0))
=> (setq x (setf y 0))
```

In this example, specifically note that the subforms of the macro expansion are not expanded; that is, the inner `setf` is still present.

macroexpand-all *form* &optional *local-macros-environment* Function

This function expands all macro calls in *form*, including those that are its subforms, and returns the result. This function knows the syntax of all Lisp special forms, so the result is completely accurate. Note, however, that any quoted list structure within *form* is not altered; there is no way to know whether you intend such list structure to be code or to be used in constructing code.

Using the `setf2` example from the **macroexpand** description, consider the following:

```
(macroexpand-all '(setf2 x y 0))
=> (setq x (setq y 0))
```

In this example, note that the inner `setf` is expanded to its corresponding `setq` form.

parse-body *body local-macros-environment documentation-allowed-p* *[c]* Function

This function parses *body* into three parts: a list of the **declare** forms, the documentation string if provided (or **nil** if not), and the remainder of the *body*. These are returned as three values in the following order: remainder of *body*, **declare** forms, and documentation. If macro expansions are necessary, they are performed using *local-macros-environment*. (Recall that macros can expand into **declare** forms if they syntactically occur where a **declare** is allowed.)

If *documentation-allowed-p* is true, a documentation string is parsed and returned. Otherwise, a string in *body* terminates the parsing of additional declarations. The &body lambda-list keyword provides a convenient way to access each of these values from a macro definition.

macroexpand-hook *[c]* Variable

The value of this variable is a function that is used by **macroexpand-1** to invoke the expansion function of a macro. It receives the same arguments as does **funcall**: the expansion function and the arguments for it.

In fact, the default value of this variable *is* **funcall**. The variable exists so that you can set it to another function, which performs the **funcall** and possibly other associated record-keeping.

The ***macroexpand-hook*** variable is not used when a macro is expanded by the interpreter.

The following function can be used as an aid for debugging macros because it prints the expansion form of an evaluated macro.

mexp &optional *form* Function

This function enters a loop in which it reads forms and iteratively expands them, printing out the result of each expansion. When the form itself has been expanded until it is no longer a macro call, **macroexpand-all** is used to expand all its subforms, and the result is printed if it is different from what preceded.

If the *form* argument is supplied, then **mexp** returns after printing the expansion. If *form* is not supplied, **mexp** prompts for a form and keeps prompting for new ones until you exit **mexp** by typing an atom or by pressing the ABORT key.

Suppose that you type the following:

```
(mexp)
```

The following prompt appears:

```
Macro-form ->
```

Next, you type the following:

```
(rest (first x))
```

Then, **mexp** prints the following:

```
(cdr (first x)) →
(cdr (car x)))
```

Note that this example demonstrates two levels of macroexpansion.

FLAVORS

19

Flavor Terminology

19.1 The flavor system is an object-oriented programming facility of the Explorer system. In essence, when you define a flavor, you define a data type and a set of operations implemented by function objects called *methods* that operate on that data type. The data type defines a set of state variables that record the unique qualities for this data type. The data type **instance** refers to data objects generated from these flavor definitions, and their associated state variables.

A flavor instance has the special quality of being **funcallable**. When this instance is applied to a set of arguments, the first argument (called the *message*) signifies a particular operation from the set of predefined methods for this flavor. This procedure is usually called *sending a message* to an object. The instance variables of that object are made available as part of the executing environment of that operation.

Mixing Flavors

19.2 The distinguishing feature of the flavor system (and the historical reasoning behind its name) comes from the ability to combine various flavor definitions to construct a new flavor, which inherits features from its component parts. The following is a simple example:

- Define a flavor called `vanilla-ice-cream`. Instance variables for such a flavor might be cream content, temperature, portion size, and so on. An operation on this flavor could be the `:eat` operation; suppose that the value returned by `:eat` is a pleasure index.

- Define another flavor called `chocolate-sauce` with instance variables cocoa content, sugar content, temperature, and portion size.

At this point, you can define a new flavor that is a combination of these two basic flavors, `ice-cream-with-chocolate-sauce`. This new flavor uses the existing definitions of its *component flavors* (in this case, `vanilla-ice-cream` and `chocolate-sauce`). It can also add its own instance variables and operations. Furthermore, it has the ability to mask part of the component flavor definitions. For example, you should define a new `:eat` operation that comprehends the benefits of chocolate sauce. By doing this, you ensure that both ice cream dishes (with and without sauce) can be properly appreciated with the now generic ice cream operation `:eat`.

Note that in this example the component flavors of `ice-cream-with-chocolate-sauce` (that is, `vanilla-ice-cream` and `chocolate-sauce`) can themselves be made up of other component flavors, thus creating a tree structure of component flavors.

Ordering Component Flavors

19.2.1 Sometimes the term *components* is used to mean the immediate components (the ones listed in the **defflavor**), and sometimes it means all the components (including the components of the immediate components, and so on). Actually, this structure is not strictly a tree, because some flavors can be components through more than one path. It is really a directed graph; it can even be cyclic.

The order in which the components are combined to form a flavor is important. The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk through of the tree, including nonterminal nodes *before* the subtrees that they head and ignoring any flavor that has been encountered previously somewhere else in the tree. This is called the *flavor precedence list*. For example, if immediate components of `flavor-1` are `flavor-2` and `flavor-3`, if the immediate components of `flavor-2` are `flavor-4` and `flavor-5`, and if the immediate component of `flavor-3` is `flavor-4`, then the complete list of components of `flavor-1` is the following:

```
flavor-1, flavor-2, flavor-4, flavor-5, flavor-3
```

The flavors earlier in this list are the more specific, less basic ones. A flavor is always the first in the list of its own components. Notice that `flavor-4` does not appear twice in this list. Only the first occurrence of a flavor appears; duplicates are removed. (The elimination of duplicates is completed during the walk; if there is a cycle in the directed graph, it does not cause a nonterminating computation.)

Instance Variables

19.2.2 The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If both `flavor-2` and `flavor-3` have instance variables named `foo`, then `flavor-1` has an instance variable named `foo`, and any methods that refer to `foo` refer to this same instance variable. Thus, different components of a flavor can communicate with one another using shared instance variables. (Typically, only one component ever sets the variable; the others only look at it.) The default initial value for an instance variable comes from the first component flavor to specify one.

Instance variable values are referenced and defined by using symbols names, and thus package qualifiers may be appropriate. Quite often, flavors defined in different namespaces are mixed together. The mechanism for referencing the correct instance variable is exactly the same as normal symbol-name resolution. For more details, see Section 5, Packages.

Primary Method

19.2.3 The way that the methods of the components are combined is the heart of the flavor system. When a flavor is defined, a single function, called a *combined method*, is constructed for each operation supported by the flavor. This function is constructed out of all the methods for that operation from all the components of the flavor. Methods can be combined in many different ways, which you can select when you define a flavor. You can also create new forms of combination.

There are several kinds of methods. The `:eat` operation that was discussed earlier defined a *primary* method. The primary method is determined by traversing the flavor precedence list and using the first untyped method for each operation name. If none is found, the list is scanned for **:default** type methods of the same name. Thus, if you are starting with a flavor `foo` and building flavor `bar` on top of it, then you can override a method of `foo` by providing your own method. Your method is invoked when it is called, leaving dormant the method for the same operation of `foo`.

Daemon Methods

19.2.4 Simple overriding is often useful; if you want to make a new flavor `bar` that is exactly like `foo`, except that it reacts completely differently to a few operations, then overriding is the appropriate procedure. However, you often do not want to completely override the base flavor's `foo` method; sometimes you want to add some extra operations to be performed. This situation calls for combining methods.

Usually, methods are combined when one flavor provides a primary method and other flavors provide *daemon methods*. Conceptually, the primary method is in charge of the main business of handling the operation, but other flavors merely want to keep informed that the message was sent or merely want to perform the part of the operation associated with their own area of responsibility. Unlike primary methods, where all but the newest definitions are ignored, all daemon methods become part of the combined operation. Daemon methods come in two kinds: *before* and *after*.

When a message is sent, it is handled by a new function called the *combined* method. The combined method first calls all of the before-daemons, then the primary method, then all the after-daemons. Each method is passed the same arguments that the combined method was given. The returned values from the combined method are the values returned by the primary method; any values returned from the daemons are ignored. The before-daemons are called in the order that flavors are combined, whereas the after-daemons are called in the reverse order. In other words, if you build `bar` on top of `foo`, then the before-daemons of `bar` run before any of those in `foo`, and the after-daemons of `bar` run after any of those in `foo`.

To approach this subject more practically, consider a simple example that is easy to manipulate, the **:print-self** method. The Lisp printer prints instances of flavors by sending them **:print-self** messages. The first argument to the **:print-self** operation is a stream, and the receiver of the message is supposed to put its printed representation on the stream. By default, flavor definitions are automatically built on top of a very basic flavor called **vanilla-flavor**. The **:print-self** method of **vanilla-flavor** actually performs the printing. Now, if you give `vanilla-ice-cream` its own primary method for the **:print-self** operation, then that method completely takes over the job of printing; the method for **vanilla-flavor** is not called at all. However, if you give `vanilla-ice-cream` a before-daemon method for the **:print-self** operation, then it is invoked before the **vanilla-flavor** method, and so whatever it prints appears before what **vanilla-flavor** prints. Thus, you can use before-daemons to add prefixes to a printed representation; similarly, after-daemons can add suffixes.

There are other ways to combine methods besides using daemons, but this way is the most common. The more advanced ways of combining methods are explained later. The **vanilla-flavor** and what it does for you are also explained later; see paragraph 19.8, **vanilla-flavor**.

Flavor Families

19.3 The following organizational conventions are recommended for all programs that use flavors.

A *base flavor* is a flavor that defines a whole family of related flavors, all of which have this flavor as one of their components. Typically, the base flavor includes features relevant to the whole family, such as instance variables, **:required-methods** and **:required-instance-variables** declarations, default methods for certain operations, **:method-combination** declarations, and documentation about the general protocols and conventions of the family. Some base flavors are complete and can be instantiated, but most are not instantiatable and merely serve as a base upon which to build other flavors. The base flavor for the *foo* family is often named *basic-foo*.

A *mixin flavor* is a flavor that defines one particular feature of an object. A mixin cannot be instantiated because it is not a complete description. Each module or feature of a program is defined as a separate mixin; a usable flavor can be constructed by choosing the mixins for the desired characteristics and combining them along with the appropriate base flavor. By organizing your flavors in this way, you keep separate features in separate flavors, and you can pick and choose among them. Sometimes the order of combining mixins does not matter, but often it does because the order of flavor combination controls the order in which daemons are invoked and wrappers are wrapped. Such order dependencies should be documented as part of the conventions of the appropriate family of flavors. A mixin flavor that provides the *mumble* feature is often named *mumble-mixin*.

If you are writing a program that uses someone else's facility to perform an operation (using that facility's flavors and methods), your program can still define its own flavors in a simple way. The facility provides a base flavor and a set of mixins: the caller can combine these in various ways, depending on exactly what it wants, because the facility probably does not provide all possible useful combinations. Even if your private flavor has exactly the same components as a preexisting flavor, it can still be useful because you can use its **:default-init-plist** (see paragraph 19.5, **defflavor** Options) to select options of its component flavors, and you can define one or two methods to perform minor customization on it.

Flavor Functions

19.4 The following are functional descriptions for flavors.

defflavor *flavor-name* ({*var*}*) ({*component*}*) {*option*|(*option* {*arg*}*)}* Macro

This macro defines a flavor. The argument *flavor-name* is a symbol that serves to name this flavor. Note that flavor names are essentially in a namespace separate from the one for functions and variables. Specifically, you can have a variable or function definition of the same name as a flavor without any ambiguity on the system's part. The call (**type-of** *obj*), where *obj* is an instance of the flavor named *flavor-name*, returns the symbol *flavor-name*. The call (**typep** *obj flavor-name*) is true if *obj* is an instance of a flavor, one of whose components (possibly itself) is *flavor-name*.

The *var* variables are the names of the instance variables containing the local state for this flavor. A list containing the name of an instance variable and a default initialization form is also acceptable; the initialization form is evaluated when an instance of the flavor is created, if no other initial value for the variable is obtained. If no initialization is specified, the variable remains unbound.

The *component* arguments are the names of the component flavors out of which this flavor is built. The features of these flavors are inherited as described previously. The components do not have to be defined before the **defflavor**, only before the flavor is instantiated.

The *option* arguments are options; each option can be either a keyword symbol or a list of a keyword symbol and its arguments. The options to **defflavor** are described later in paragraph 19.5, **defflavor** Options. For an example of **defflavor**, see **make-instance**.

defmethod (*flavor-name* [*method-type*] *operation*) *lambda-list* {*declaration*|*doc-string*}* {*body-form*}* Macro

defmethod (*flavor-name* [*method-type*] *operation*) *function* Macro

This macro defines a method that is a function that handles a particular operation for instances of a particular flavor. The *flavor-name* argument is a symbol that is the name of the flavor that is to receive the method. The *operation* argument is a symbol that names the operation to be handled. It is usually a keyword, so it can be used easily from any package. The *method-type* argument is a keyword symbol for the type of method. It is omitted when you are defining a primary method. For some method types, additional information is expected after the *operation*.

The meaning of the *method-type* depends on what kind of method combination is declared for this operation. For instance, **:before** and **:after** are allowed for daemons (paragraph 19.6, Method Combination Type).

The *lambda-list* describes the parameters and &aux variables of the function. The first argument to the method, which is the operation name itself, is automatically handled and thus is not included in the lambda list. Note that methods cannot have unevaluated (&quote) arguments; that is, they must be functions, not special forms. The *body-form*s are the function body; the value of the last form is returned if this is a primary method.

The variant form for this macro is the following:

(**defmethod** (*flavor-name operation*) *function*)

The *function* argument is an unquoted symbol that names a function and whose functional definition is the implementation of the operation for the flavor of *flavor-name*. This function must take appropriate arguments, the first being the operation name and the rest, if any, being the arguments for the operation. The function should be defined by **defun-method**.

If you redefine a method that is already defined, the old definition is replaced by the new one. Given a flavor, an operation name, and a method type, there can only be one function (with the exception of **:case** methods), so if you define a **:before** daemon method for the `foo` flavor to handle the `:bar` operation, then you replace the previous before-daemon of the flavor `foo`; however, you do not affect the primary method or methods of any other type, operation, or flavor.

Note that the compiler can optimize the combined methods into one function call; that is, daemon methods are compiled inline when speed is more important than safety, thus avoiding function calling overhead. However, if daemon methods for component flavors are redefined, it is necessary to recompile the flavor methods to receive the benefit of the new daemon method. See Section 21, Compiler Operations, for more details.

The function specification for a method resembles the following:

(**:method** *flavor-name operation*)
or:
(**:method** *flavor-name method-type operation*)
or:
(**:method** *flavor-name method-type operation suboperation*)

Note that the *flavor-name* is actually a symbol and requires you to supply a package prefix when the symbol is not in your namespace. The other items in the specification are keywords. None of the items in the function spec need to be quoted. This specification is useful to know if you want to use Edit Definition in Zmacs by invoking **trace**, **break**, or **advise** on a method, or if you want to experiment with the method function itself.

For an example of **defmethod**, see the example for **make-instance**.

make-instance *flavor-name* {*init-option value*}* Function

This function creates and returns an instance of the specified flavor. Arguments after the *flavor-name* are alternating initialization option keywords and arguments to those keywords. These options are used to initialize instance variables and to select arbitrary options. Instance variables are initialized by supplying the instance variable name, written as a keyword, followed by the desired initial value. An **:init** message is sent to the newly created object with one argument—the initialization property list. This is a property list containing the initialization options specified on the **make-instance** call and those defaulted from the flavor's **:default-init-plist** option (however, initialization keywords that simply initialize instance variables, and the corresponding values, can be absent when the **:init** methods are called). The **make-instance** function is an easy-to-call interface to **instantiate-flavor**.

Consider the following example:

```
(defflavor vanilla-ice-cream
           (cream-content                ; instance variables
            ice-cream-temp
            ice-cream-portion)
           ()                            ; no component flavors
  :settable-instance-variables)  ; defflavor options

(defmethod (vanilla-ice-cream :eat) ()  ; define the :EAT operation
  (typecase ice-cream-temp              ; returned value is pleasure index
    ((integer 20 32) (* ice-cream-portion cream-content))
    (t (- (* ice-cream-portion cream-content)))))

(setf dessert (make-instance 'vanilla-ice-cream   ; dish up the ice cream
                             :cream-content 10
                             :ice-cream-temp 30
                             :ice-cream-portion 8))

(send dessert :eat) => 80               ; mmmm!
```

all-flavor-names Variable

This variable is a list of the names of all the flavors that have ever been defined.

instantiate-flavor *flavor-name init-plist* &optional *send-init-message-p return-unhandled-keywords-p area* Function

This function is an extended version of **make-instance**, providing more features. Note that it takes the *init-plist* as an argument, rather than taking an unspecified number of arguments containing initialization options and values.

The *init-plist* argument must be a disembodied property list (that is, suitable as an argument to **get** rather than **getf**); **locf** of a &rest argument can be used for this argument. However, this property list can be modified. The properties from the **default-init-plist** are added by **putprop** if not already present, and some **:init** methods perform explicit **putprop** calls onto the *init-plist*.

Do not use **nil** as the *init-plist* argument, which would mean using the properties of the symbol **nil** as the initialization options. If your goal is to have no initialization options, you must provide a property list containing no properties, such as `(nil)`, a list with a single element of `nil`.

The **:init** methods must not look on the *init-plist* for keywords that simply initialize instance variables (that is, keywords defined with **:inittable-instance-variables** rather than with **:init-keywords**). The corresponding instance variables are already set up when the **:init** methods are called, and sometimes the keywords and their values can actually be missing from the *init-plist* if it is more efficient not to include them. To avoid problems, always refer to the instance variables themselves rather than looking for the keywords that initialize them.

In the event that **:init** methods perform a **remprop** operation on the properties already on the initialization property list (as opposed to simply executing **get** and **putprop**), this property list is destructively modified. Thus, the original list of options is modified.

When an instance is created the flavor's methods, instance variables, and other internal data are collected and validated according to the flavor specifications. This happens once for each instantiated flavor unless some part of the flavor is redefined, in which case it must be recomputed on the first subsequent instantiation. This procedure can take a lot of time and can invoke the compiler, but it happens only once for a particular flavor no matter how many instances you make, unless you redefine the flavor or one of its methods.

Next, the instance variables are initialized. This initialization can happen in several ways. If an instance variable is declared inittable and a keyword with the same spelling as its name appears in *init-plist*, it is set to the value specified after that keyword. If an instance variable is not initialized in this way and an initialization form is specified for it in a **defflavor**, that form is evaluated and the variable is set to the result. The initialization form cannot refer to any instance variables or to **self**; it is not evaluated in the *inside* environment from which methods are called. If an instance variable is not initialized in either of these ways, it is left unbound; an **:init** method can initialize it.

If any keyword appears in the *init-plist* but is not used to initialize an instance variable and is not declared in an **:init-keywords** option, it is presumed to be misspelled. So any keywords that you handle in an **:init** handler should also be mentioned in the **:init-keywords** option of the definition of the flavor.

If the *return-unhandled-keywords-p* argument is not supplied, the system complains about such keywords by signaling an error. But if *return-unhandled-keywords-p* is a true value, a list of such keywords is returned as the second value of **instantiate-flavor**.

If the *send-init-message-p* argument is supplied and is true, an **:init** message is sent to the newly created instance with one argument—the *init-plist*. You can use **get** to extract options from this property list. Each flavor that needs initialization can contribute to the **:init** operation by defining a daemon method.

If the *area* argument is specified, it is the number of an area in which the instance is to be allocated; otherwise, it is allocated in the default area.

:init *init-plist* — Method of *all flavors*

This default primary method does nothing. It exists primarily to allow daemon method definitions. The *init-plist* argument is actually a locative to the initialization property list generated by **instantiate-flavor**. The *init-plist* argument is passed to the combined **:init** method. Use **get** to examine the values of any particular option.

By convention, never define a primary **:init** method of your own. You should always assume that another flavor mixes in your flavor to define a more meaningful object. If primary **:init** methods are defined by component flavors, they are all masked out. Therefore, use **:before** and **:after** methods to initialize instances.

instancep *object* — Function

This function returns true if *object* is an instance. This predicate is equivalent to the following form:

(**typep** *object* **'instance**)

The following forms and variables are used to support the flavor system.

undefflavor *flavor-name* — Function

This function undefines the flavor specified by *flavor-name*. All methods of *flavor-name* are lost. You can no longer instantiate *flavor-name* or any flavors that depend on it.

If instances of the discarded definition exist, they continue to use that definition.

undefmethod (*flavor-name* [*method-type*] *operation* [*suboperation*]) — Macro

This macro removes a method of a flavor. Consider the following form:

(`undefmethod` (*flavor-name* `:before` *operation*))

This form removes the **:before** method created by the following:

(`defmethod` (*flavor-name* `:before` *operation*) (*args*)...)

To remove a wrapper, use **undefmethod** with **:wrapper** as the method type. The **undefmethod** macro is simply an interface to **fundefine** that accepts the same syntax as **defmethod**.

If a file that formerly contained a method definition is reloaded and if that method no longer seems to have a definition in the file, you are asked whether to perform an **undefmethod** operation on that method. This operation can enable the modified program to inherit the methods it is supposed to inherit. If the method in question has been redefined by another file, this operation is not performed, the assumption being that the definition was merely moved.

self Variable

When a message is sent to an object, this variable is automatically bound to that object for the benefit of methods that want to manipulate the object itself (as opposed to its instance variables).

funcall-with-mapping-table *function mapping-table* &rest *arguments* Special Form

With this special form, the argument *function* is applied to *arguments* with **sys:self-mapping-table** bound to *mapping-table*, which is not evaluated. This procedure is faster than binding the variable yourself and performing an ordinary **funcall**, because the system assumes that the mapping table that you specify is the correct one for running *function*. However, if you pass the wrong mapping table, the function executes incorrectly. (For more information about mapping tables, see paragraph 19.9, Property List Operations.)

This function is used in the code for combined methods and is also useful in **:around** methods (see Method Combination Type, paragraph 19.6).

lexpr-funcall-with-mapping-table *function mapping-table* {*argument*}* *arglist* Special Form

With this special form, the argument *function* is applied to *arguments* using **lexpr-funcall** with **sys:self-mapping-table** bound to *mapping-table*, which is not evaluated. (For more information about mapping tables, see paragraph 19.9, Property List Operations.)

defun-method *function-spec flavor-name lambda-list* {*body-form*}* Macro

This macro allows you to write forms that look like functions yet have access to the instance variables for a particular object. The caller of these functions should be a method, or another **defun-method** of *flavor-name* or of some other flavor that has *flavor-name* as a component flavor. (Specifically, the value of **self** in the calling environment must be bound to an appropriate flavor instance.) Thus, if your methods become unmaintainable due to their size, you can break them into modules by using **defun-methods**.

The **defun-method** macro is equivalent to the following:

```
(defun function-spec lambda-list
  (declare (:self-flavor flavor-name))
   . body)
```

If the compiler sees a particular **defun-method** before any references to it, the compiler can generate the more efficient call form **funcall-with-mapping-table**. If the reference is seen first, then the instance-variable mapping table must be looked up at run time.

with-self-variables-bound {*body-form*}* Special Form

Within the body of this special form, all of the instance variables of **self** are bound as specials to the values inside **self**. (Without this special form, this convention holds only for those instance variables that are specified in **:special-instance-variables** when the flavor of **self** is defined.) As a result, inside the body you can freely use **set**, **boundp**, and **symbol-value** on the instance variables of **self**.

This special form is used by the interpreter when an uncompiled method is executed so that the interpreted references to instance variables work properly.

declare-flavor-instance-variables (*flavor-name*) {*body-form*}* Macro

When you wrap this macro around a function definition, it allows you to define a function that is not itself a method but that is to be called by methods and wants to be able to access the instance variables of the object **self**. The following form surrounds the function definition with a peculiar kind of declaration that makes the instance variables of `flavor` accessible by name:

```
(declare-flavor-instance-variables (flavor)
  (defun function args body...))
```

With this form, any kind of function definition is allowed; it does not have to use **defun**.

If you call such a function when **self**'s value is an instance whose flavor does not include `flavor` as a component, an error is signaled.

However, using a local declaration is cleaner than using **declare-flavor-instance-variables** because local declarations do not involve putting anything around the function definition. Put (**declare** (**:self-flavor** *flavor-name*)) as the first expression in the body of the function. For example:

```
(defun foo (a b)
  (declare (:self-flavor my-flavor))
  (+ a (* b speed)))
```

In this example, `speed` is an instance variable of the flavor `my-flavor`. The following form is equivalent to the preceding:

```
(declare-flavor-instance-variables (my-flavor)
(defun foo (a b)
  (+ a (* b speed))))
```

recompile-flavor *flavor-name* &optional *single-operation* *use-old-combined-methods-p do-dependents-p* Function

This function updates the internal data of the flavor and any flavors that depend on it. If supplied, *single-operation* should be an operation name, in which case only the methods for that operation are changed.

The system follows this procedure when you define a new method that did not previously exist. If *use-old-combined-methods-p* is true, then the existing combined method functions are used if possible. New functions are generated only if the set of methods to be called has changed. The default value for this argument is true. If *use-old-combined-methods-p* is **nil**, automatically generated functions for calling multiple methods or for containing code generated by wrappers are regenerated unconditionally. If *do-dependents-p* is **nil** (the default is true), only the flavor you specified is recompiled. Normally, it and all flavors that depend on it are recompiled.

The **recompile-flavor** function affects only flavors that have already been compiled. Typically, this means that it affects flavors that have been instantiated but does not bother with mixins.

sys:*dont-recompile-flavors* Variable

If this variable is true, combined methods are not automatically recompiled.

If you wish to make several changes, each of which will cause recompilation of the same combined methods, you can use this variable to speed up processing by forcing the recompilations to happen only once. Set the variable to true, make your changes, and then set the variable back to **nil**. Then use **recompile-flavor** to recompile any combined methods that need recompilation. For example:

```
(setf sys:*dont-recompile-flavors* t)
(undefmethod (w:sheet :after :bar))
(defmethod (w:sheet :before :bar) ...)
(setf sys:*dont-recompile-flavors* nil)
(recompile-flavor 'w:sheet :bar)
```

Since `w:sheet` has many dependents and `recompile-flavor` takes quite a long time to execute, this procedure can save you considerable time.

compile-flavor-methods *{flavor-name}** Macro

When placed in a file to be compiled, this form causes the compiler to include the automatically generated combined methods for the named flavors in the resulting object file, provided that all of the necessary flavor definitions have been made. Furthermore, when the object file is loaded, internal data structures (such as the list of all methods of a flavor) are generated.

Thus, combined methods are compiled at compile time, and the data structures are generated at load time, rather than both operations happening at run time. This procedure is very advantageous, because if the compiler must be invoked at run time, the program is slowed the first time it is run. (The compiler is still called if incompatible changes have been made, such as addition or deletion of methods that must be called by a combined method.)

Use **compile-flavor-methods** only for flavors that are going to be instantiated by name. For a flavor that is never instantiated (that is, a flavor that serves only as a component of other flavors that actually are instantiated), using this macro is a complete waste of time.

The **compile-flavor-methods** forms should be compiled after all of the information needed to create the combined methods is available. Put these forms after all of the definitions of all relevant flavors, wrappers, and methods of all components of the flavors mentioned.

The methods used by **compile-flavor-methods** to form the combined methods that go in the object file are all those present in the Lisp environment at that time.

When a **compile-flavor-methods** form is seen by the interpreter, the combined methods are compiled and the internal data structures are generated.

get-handler-for *object operation* Function

Given an object and an operation, this function returns the object's method for the operation, or **nil** if it has none. When *object* is an instance of a flavor, this function can be useful to find which of that flavor's components supplies the method. If a combined method is returned, you can use the Zmacs command List Combined Methods to find out what it does.

This function can be used with features other than flavors and has an optional argument that is not relevant here and not documented.

flavor-allows-init-keyword-p *flavor-name keyword* Function

This pseudo-predicate returns **nil** if the flavor named *flavor-name* does not allow *keyword* in the initialization options when it is instantiated. Otherwise, this function returns the name of the component flavor that contributes the support of that keyword.

sys:flavor-allowed-init-keywords *flavor-name* Function

This function returns a list of all the initialization keywords that can be used in instantiating *flavor-name*.

symeval-in-instance *instance symbol* &optional *no-error-p* Function

This function is used to find the value of an instance variable inside a particular instance. The argument *instance* is the instance to be examined, and *symbol* is the instance variable whose value is returned. Note the the *symbol* argument may need a package prefix if the corresponding symbol (the one referenced in the **defflavor**) is not in the current symbol namespace. If there is no such instance variable, an error is signaled unless *no-error-p* is true, in which case **nil** is returned.

set-in-instance *instance symbol value* Function

This function is used to alter the value of an instance variable inside a particular instance. The argument *instance* is the instance to be altered; *symbol* is the instance variable whose value is set; and *value* is the new value. If there is no such instance variable, an error is signaled. Note the the *symbol* argument may need a package prefix if the corresponding symbol (the one referenced in the **defflavor**) is not in the current symbol namespace. This function is equivalent to **setf** of **symeval-in-instance**.

locate-in-instance *instance symbol* Function

This function returns a locative pointer to the cell inside *instance* that holds the value of the instance variable named *symbol*. Note that the *symbol* argument may need a package prefix if the corresponding symbol (the one referenced in the **defflavor**) is not in the current symbol namespace. This function is equivalent to **locf** of **symeval-in-instance**.

describe-flavor *flavor-name* Function

This function prints out descriptive information about a flavor; it is self-explanatory. Fortunately, it provides a combined list of component flavors. This list is what is printed after the phrase *and directly or indirectly depends on*.

sys:*flavor-compilations* Variable

This variable contains a history of when the flavor mechanism invoked the compiler. The value of this variable is a list. Elements toward the front of the list represent more recent compilations. Elements are typically of the form (*function-spec pathname*), where the function specification is a list that starts with **:method** and has a method type of **:combined**.

It is sometimes useful to set this variable to **nil** before loading files that you suspect may have missing or obsolete **compile-flavor-methods** in them. After loading the file, do whatever is needed to create all the required instances; then examine this list.

defflavor Options

19.5 Several of the following options declare things about instance variables. These options can be given with arguments that are instance variables or without any arguments, in which case they refer to all of the instance variables listed at the top of the **defflavor**. The affected instance variables do not include those of the component flavors. When arguments are given, they must be instance variables that are listed at the top of the **defflavor**; otherwise, they are assumed to be misspelled, and an error is signaled. You can declare things about instance variables inherited from a component flavor, but to do so you must list these instance variables explicitly in the instance variable list at the top of the **defflavor**.

:gettable-instance-variables — This option enables automatic generation of methods for retrieving the values of instance variables. The operation name is the name of the variable in the KEYWORD package (that is, with a colon in front of it).

Note that there is nothing special about these methods; you can easily define them yourself. This option causes **defflavor** to generate them automatically to save you the trouble of writing out numerous very simple method definitions. If you define a method with the same operation name as one of the automatically generated methods, your definition overrides the automatically generated one, just as if you had manually defined two methods for the same operation name.

:inittable-instance-variables — With this option, the instance variables listed as arguments are made *inittable*. This means that they can be initialized through use of a keyword (a colon followed by the name of the variable) as an initialization option argument to **make-instance**.

:settable-instance-variables — This option enables automatic generation of methods for setting the values of instance variables. The operation name is **:set-** followed by the name of the variable. All settable instance variables are also automatically made gettable and inittable. (See the note in the description of the **:gettable-instance-variables** option.)

In addition, **:case** methods are generated for the **:set** operation with suboperations taken from the names of the variables so that the **:set** operation can be used on them.

:special-instance-variables — With this option, the instance variables listed as arguments are made special. Whenever a message is sent to an instance of this flavor (or any containing flavor), these instance variables are actually bound as specials: they are bound through the execution of all the methods.

You must use this option to identify any instance variables that you wish to be accessible through **symbol-value**, **set**, **boundp**, and **makunbound**.

Because these functions refer only to the special value cell of a symbol, values of instance variables not made special are not visible to them.

Use this option for any instance variables that are declared globally special. If you do not do so, the flavor system does it for you automatically when you instantiate the flavor and gives you a warning to remind you to fix the **defflavor**.

:init-keywords — With this option, the arguments are declared to be valid keywords to supply as arguments to **instantiate-flavor** or **make-instance** when creating an instance of this flavor (or any flavor containing it). The system uses this for error checking. Before the system sends the **:init** message, it makes sure that all the keywords in the init-plist are either inittable instance variables or elements of this list. If the caller misspells a keyword or otherwise uses a keyword that no component flavor handles, this feature will signal an error. Therefore, if you write an **:init** method that processes a keyword (one that does not correspond to an instance variable), then this keyword should be listed in the **:init-keywords** option of the flavor.

:default-init-plist — With this option, the arguments are alternating keywords and value forms, like a property list. When the flavor is instantiated, these properties and values are put into the init-plist unless already present. This allows one component flavor to default an option to another component flavor. The value forms are only evaluated when and if they are used. For example:

```
(:default-init-plist :frob-array (make-array 100))
```

This form provides a default `frob-array` for any instance in which you do not provide one explicitly.

:included-flavors — With this option, the arguments are names of flavors to be included in this flavor. The difference between declaring flavors here and declaring them at the top of the **defflavor** is that when component flavors are combined, if an included flavor is not specified as a normal component, it is inserted into the list of components immediately after the last component to include it. Thus, **:included-flavors** ensures that the flavor is included. If an included flavor *is* specified as a component, its position in the list of components is completely controlled by that specification, independently of where in the list the flavor that includes it appears.

The options **:included-flavors** and **:required-flavors** are used in similar ways. The difference is that when a flavor is required but not given as a normal component, an error is signaled, but when a flavor is included but not given as a normal component, it is automatically inserted into the list of components at a *reasonable* place.

:no-vanilla-flavor — If any component of a flavor specifies this option, then **sys:vanilla-flavor** is not included in that flavor. This option should not be used casually.

Normally, when a flavor is instantiated, the special flavor **sys:vanilla-flavor** is included automatically at the end of its list of components. The vanilla flavor provides default methods for the standard operations that all objects are supposed to understand. These include **:print-self**, **:describe**, **:which-operations**, and several other operations. (See paragraph 19.8, **vanilla-flavor**.)

:default-handler — With this option, the argument is the name of a function that is to be called when a message is received for which there is no method. This function is called with whatever arguments the instance was called with, including the operation name; whatever values the function returns are returned by this operation. If this option is not specified for any component flavor, it defaults to a function that signals an error.

:ordered-instance-variables — This option is mostly for esoteric internal system uses. The arguments are names of instance variables that must appear first (and in this order) in all instances of this flavor or of any flavor depending on this flavor. This option is used for instance variables that are specially known by microcode and in connection with the **:outside-accessible-instance-variables** option. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this **defflavor**.

Removing any of the **:ordered-instance-variables** or changing their positions in the list requires that you recompile all methods that use any of the affected instance variables.

:outside-accessible-instance-variables — The arguments are instance variables that are to be accessible from *outside* of this object, that is, from functions other than methods. A macro is defined that takes an object of this flavor as an argument and returns the value of the instance variable; **setf** can be used to set the value of the instance variable. The name of the macro is the name of the flavor concatenated with a hyphen and the name of the instance variable. These macros are similar to the accessor macros created by **defstruct**. (See the following description of **:accessor-prefix**.)

This feature works in two different ways, depending on whether the instance variable has been declared to have a fixed slot in all instances via the **:ordered-instance-variables** option.

If the variable is not ordered, the position of its value cell in the instance must be computed at run time. This computation takes noticeable time, although less than for sending a message. An error is signaled if the argument to the accessor macro is not an instance or is an instance that does not have an instance variable with the appropriate name. However, there is no error checking to determine if the flavor of the instance is the flavor that the accessor macro was defined for or is a flavor built upon that flavor.

If the variable is ordered, a call to the accessor macro is compiled into a subprimitive that simply accesses by number this variable's assigned slot. This subprimitive is only three or four times slower than **car**. The only error checking performed is to make sure that the argument is actually an instance and is actually big enough to contain that slot. There is no check to determine if the accessed slot actually belongs to an instance variable of the appropriate name.

Note that if **:ordered-instance-variables** has changed, this method of accessing instance variables cannot be used to look at old instances of flavors because the slot number may be different.

:accessor-prefix — This option allows you to specify the accessor prefix for the macros created by the **:outside-accessible-instance-variables** option. The argument to this option should be a symbol. Normally, the accessor macro created by the **:outside-accessible-instance-variables** option to access (for instance, the `flv` flavor's instance variable `x`) is named `flv-x`. Specifying `(:accessor-prefix get)` causes it to be named `getx` instead.

:alias-flavor — This option marks the flavor as being an alias for another flavor. This flavor should have only one component, which is the flavor it is an alias for, and no instance variables or other options. No methods should be defined for it.

The effect of the **:alias-flavor** option is that an attempt to instantiate this flavor actually produces an instance of the other flavor.

The alias flavor and its base flavor are also equivalent when used as an argument of **subtypep** or as the second argument of **typep**. However, if the alias status of a flavor is changed, you must recompile any code that uses it as the second argument to **typep** in order for such code to work properly.

The **:alias-flavor** option is mainly used for changing a flavor's name conveniently.

:method-combination — This option declares the way that methods from different flavors are combined. Each argument to this option is a list of the following format:

(*combination-type order operation1 operation2...*)

The variables *operation1*, *operation2*, and so forth, are names of operations whose methods are to be combined in the declared fashion. The element specified by *combination-type* is a keyword that is a defined type of method combination. The element specified by *order* is a keyword whose interpretation is left up to *combination-type*; typically, it is either **:base-flavor-first** or **:base-flavor-last**.

Any component of a flavor can specify the type of method combination to be used for a particular operation. If no component specifies a type of method combination, then the default type is used, namely **:daemon**. If more than one component of a flavor specifies the type of method combination, then they must agree on the specification or else an error is signaled. Method combination is discussed in paragraph 19.6, Method Combination Type.

:instance-area-function — The argument to this option is the name of a function to be used when this flavor is instantiated, to determine in which area to create the new instance. Use a function name rather than an explicit lambda expression. For example:

(**:instance-area-function** *function-name*)

When the instance area function is called, it is given the initialization property list as an argument and should return an area number or **nil** to use the default. Initialization keyword values can be accessed by using **get** on the initialization property list.

Instance area functions can be inherited from component flavors. If a flavor does not have or inherit an instance-area function, its instances are created in **default-cons-area**.

:instantiation-flavor-function — You can define a flavor `neopolitan` such that, when you try to instantiate it, it calls a function to decide what flavor it should really instantiate (not necessarily `neopolitan`). This operation is performed by giving `neopolitan` an instantiation flavor function:

(**:instantiation-flavor-function** *function-name*)

When the form `(make-instance 'neopolitan` *keyword-args* `...)` is executed, the instantiation flavor function is called with two arguments: the flavor name specified (`neopolitan` in this case) and the initialization property list (the list of keyword arguments). This instantiation flavor function should return the name of the flavor that should actually be instantiated.

Note that the instantiation flavor function applies only to the flavor for which it is specified. It is not inherited by dependent flavors.

:run-time-alternatives, :mixture — A run-time-alternative flavor defines a collection of similar flavors, all built on the same base flavor but having various mixins as well. Instantiation chooses a flavor of the collection at run time based on the initialization keywords specified, using an automatically generated instantiation flavor function. For example:

```
(defflavor spumoni () (basic-spumoni)
    (:run-time-alternatives
        (:big big-spumoni-mixin))
    (:init-keywords :big :wide))
```

After this form is executed, `(make-instance 'spumoni :big t)` makes an instance of a flavor whose components are `big-spumoni-mixin` as well as `spumoni`. But `(make-instance 'spumoni)` or `(make-instance 'spumoni :big nil)` makes an instance of `spumoni` itself. The clause `(:big big-spumoni-mixin)` in the `run-time-alternatives` specifies to incorporate `big-spumoni-mixin` if `:big`'s value is `t`, but not if it is `nil`.

There may be several clauses in the `:run-time-alternatives`. Each one is processed independently. Thus, the two keywords `:big` and `:wide` could independently control two mixins, giving four possibilities:

```
(defflavor spumoni () (basic-spumoni)
    (:run-time-alternatives
        (:big big-spumoni-mixin)
        (:wide wide-spumoni-mixin)
    (:init-keywords :big)))
```

You can test for values other than `t` and `nil`. The following clause allows the value for the keyword `:size` to be `:big`, `:small`, or `nil` (or omitted):

```
(:run-time-alternatives
   (:size (:big big-spumoni-mixin)
          (:small small-spumoni-mixin)
          (nil nil))
(:init-keywords :size)
```

If the value of `:size` is `nil` or omitted, no mixin is used (thus, the second `nil`). If this value is `:big` or `:small`, an appropriate mixin is used. This kind of clause is distinguished from the simpler kind by having a list as its second element. The values to check for can be anything, but **eq** is used to compare them.

The value of one keyword can control the interpretation of others by nesting clauses within clauses. If an alternative has more than two elements, the additional elements are subclauses that are considered only if that alternative is selected. For example, the following clause specifies to consider the `:size` keyword only if `:ethereal` is `nil`.

```
(:run-time-alternatives
   (:ethereal (t ethereal-mixin)
              (nil nil
              (:size (:big big-spumoni-mixin)
                     (:small small-spumoni-mixin)
                     (nil nil))))
(:init-keywords :size)
```

The **:mixture** option is synonymous with **:run-time-alternatives**. It exists for compatibility purposes.

:documentation — This option specifies the documentation string for the flavor definition, which is made accessible through the following form:

(documentation *flavor-name* 'flavor)

You can also display this documentation with the **describe-flavor** function (see paragraph 19.4, Flavor Functions) or the Zmacs editor's Describe Flavor command (see the Describe Flavor command in the *Explorer Zmacs Editor Reference* manual).

Some flavors are never meant to be instantiated by themselves. They are designed to be mixed in with, and are dependent upon, other flavors. These flavors are mixins and should have the **:abstract-flavor** option.

:abstract-flavor — This option marks the flavor as one that is not supposed to be instantiated (that is, is supposed to be used only as a component to other flavors). An attempt to instantiate the flavor causes an error to be signaled.

It is sometimes useful to execute **compile-flavor-methods** on a flavor that is not going to be instantiated if the combined methods for this flavor are inherited and shared by many others. The **:abstract-flavor** option tells **compile-flavor-methods** not to notice missing required flavors, methods, or instance variables. Presumably, the missing elements are supplied by the flavors that depend on this one and that are actually instantiated.

:required-instance-variables — This option declares that this flavor intends to use instance variables that are not defined in this **defflavor**. Specifically, it assumes that some component flavor, or some parent flavor, will define this instance variable. If there is an attempt to instantiate a flavor that incorporates this flavor, an error occurs if it does not have these required instance variables in its set of instance variables. Otherwise, required instance variables are normal instance variables.

:required-methods — This option declares that this flavor intends to use methods that are not defined in this **defflavor**. Specifically, it assumes that some component flavor, or some parent flavor, will define this method. An error occurs if there is an attempt to instantiate such a flavor when it is lacking a method for one of these operations. Typically, this option appears in the **defflavor** for a base flavor. Usually, this option is used when a base flavor performs a **send self** to send itself a message that is not handled by the base flavor itself; the base flavor is not instantiated alone but only with other components (mixins) that do handle the message. With this keyword the error of having no handler for the message can be detected when the flavor is instantiated or when **compile-flavor-methods** is executed, rather than when the missing operation is used.

:required-flavors — With this option, the arguments are names of flavors that any other flavor incorporating this one must include as components, directly or indirectly. The difference between declaring flavors as required and listing them directly as components at the top of the **defflavor** is that declaring flavors to be required does not make any commitments about where these flavors appear in the ordered list of components; the position of these flavors is left up to whoever does specify them as components. The purpose of declaring a flavor to be required is to allow access to instance variables declared by that flavor. It also provides error checking: an attempt to instantiate a flavor that does not include the required flavors as components produces an error. Compare this option with **:required-methods** and **:required-instance-variables**.

Method Combination Type

19.6 As was mentioned earlier, there are many ways to combine methods. The way discussed previously is called **:daemon** method combination. To use one of the others, use the **:method-combination** option to **defflavor** (paragraph 19.5, defflavor Options) to indicate that all the methods for a certain operation on this flavor, or any flavor built on it, are to be combined in a certain way.

Note that for most types of method combination other than **:daemon** you must define the order in which the methods are combined, either with **:base-flavor-first** or **:base-flavor-last** in the **:method-combination** option. In this context, *base-flavor* means the last element of the flavor's fully expanded list of components.

A few method types, such as **:default** and **:around**, have a universal meaning independent of the method combination type. Aside from these, the permitted method type keywords vary depending on the type of method combination selected, and many combination types allow only untyped methods. Certain method types are also used for internal purposes. The following are the combination types that can be specified for the **:method-combination** option of **defflavor**.

:daemon — This option is the default type of method combination. First, all the **:before** methods are called; then the primary (untyped) method for the outermost flavor is called; then all the **:after** methods are called. The value returned is the value of the primary method. This kind of method combination is available by default.

:progn — With this combination type, all the methods are called inside a **progn** special form. No typed methods (except for **:progn**) are allowed. The result of the combined method is whatever the last of the methods returns.

:or — With this combination type, all the methods are called inside an **or** special form. No typed methods (except for **:or**) are allowed. Thus, each of the methods is called in turn. If a method returns a true value, that value is returned, and none of the rest of the methods are called; otherwise, the next method is called. In other words, each method is given a chance to handle the message; if it does not want to handle the message, it returns **nil**, and the next method is allowed to try.

:and — With this combination type, all the methods are called inside an **and** special form. No typed methods (except for **:and**) are allowed. The basic idea is much like :or, which is described above.

:daemon-with-or — This option is like the **:daemon** method combination type, except that the primary method is wrapped in an **or** special form with all **:or** methods. Multiple values are returned from the primary method but not from the **:or** methods. This produces combined methods that act something like the following:

```
(progn (foo-before-method)
       (multiple-value-prog1
         (or (foo-or-method)
             (foo-primary-method))
         (foo-after-method)))
```

This is useful primarily for flavors in which a mixin introduces an alternative to the primary method. Each **:or** method is allowed to run before the primary method and decides whether the primary method is run or not; if any **:or** method returns a true value, the primary method is not run (nor are the rest of the **:or** methods).

:daemon-with-and — This option is like **:daemon-with-or**, except that it combines **:and** methods in an **and** macro. The primary method is run only if all of the **:and** methods return true values.

:daemon-with-override — This option is like the **:daemon** method combination type, except an **or** macro is wrapped around the entire combined method with all **:override** typed methods before the combined method. This differs from **:daemon-with-or** in that the **:before** and **:after** daemons are run only if *none* of the **:override** methods return a true value. The combined method resembles the following:

```
(or (foo-override-method)
    (progn (foo-before-method)
           (foo-primary-method)
           (foo-after-method)))
```

:append — With this combination type, all the methods are called, and the values are appended together. No typed methods (except for **:append**) are allowed. The **:append** methods are called first, and then the untyped methods are called.

:nconc — With this option, all the methods are called, and the values are **nconced** together. Only untyped and **:nconc** methods are allowed.

:list — This keyword calls all the methods and returns a list of their returned values. No typed methods (except for **:list**) are allowed.

:inverse-list — This combination type derives its name from its symmetric similarity with the **:list** combination type. Specifically, for **:inverse-list** methods, the caller supplies only one argument, which is a list. Each method that is called to process this operation is supplied the next element from the list as its single argument. Thus, where **:list** method combinations return a list of the values of the combined methods, **:inverse-list** method combinations supply an element from an input list as an argument to each of the combined methods. The value returned from methods combined with **:inverse-list** is undefined. The only types of methods that can be combined in this way are untyped methods and **:inverse-list** methods.

Note that because this combination type implicitly excludes **:before** and **:after** methods, the operations mentioned here are those in which each combined method is defined in a component flavor. Therefore, you can mix several flavors that have a **:list** combined method of the same method name and an **:inverse-list** combined method of a different name. Under these conditions, note that the order in which the combined methods are called is the same for both the **:list** and **:inverse-list** operations. Therefore, the list returned from a **:list** combined method could be used as the input argument to an **:inverse-list** combined method and thus be processed by the respective component flavor.

:pass-on — This combination type calls each method with the values returned by the preceding one. The values returned by the combined method are those of the outermost call. The format of the declaration in the **defflavor** is the following:

(**:method** (**:pass-on** (*ordering* {*arg*}*)) {*operation-name*}*)

In this format, *ordering* is **:base-flavor-first** or **:base-flavor-last**. The *args* should be the argument list for each of the combined methods. Note that the operation handler expects each method called to return multiple values that conform to *args*. The argument *args* can include the &aux and &optional keywords.

Only untyped or **:pass-on** methods are allowed. The **:pass-on** methods are called first.

:case — With this method combination, the combined method automatically performs a **case** dispatch on the first argument of the operation, known as the *suboperation*. Methods of type **:case** can be used, and each one specifies one suboperation to which it applies. If no **:case** method matches the suboperation, the primary method, if any, is called. For example, suppose you have the following flavor:

```
(defflavor foo (a b) ()
 (:method-combination (:case :base-flavor-last :win)))

(defmethod (foo :case :win :a) ( ) a)
```

Given these definitions, the following message is handled:

```
(send some-foo-instance :win :a)
```

This next method handles the form `(send a-foo :win :a*b)`:

```
(defmethod (foo :case :win :a*b) ()
  (* a b))
```

Finally, the following method handles the **otherwise** case, such as `(send a-foo :win :something-else)`:

```
(defmethod (foo :win) (suboperation)
  (list 'something-random suboperation))
```

The **:case** methods are unusual in that one flavor can have many **:case** methods for the same operation, as long as they are for different suboperations.

The suboperations **:which-operations**, **:operation-handled-p**, **:send-if-handles**, and **:get-handler-for** are all handled automatically, based on the collection of **:case** methods that are present.

Methods of type **:or** are also allowed. They are called just before the primary method, and if one of them returns a true value, that is the value of the operation. No more methods are called.

Method Type

19.7 The following is a list of all the method types used in the standard system. You can add more by defining new forms of method combination. If no type is given to **defmethod**, a primary method is created. This is the most common type of method.

:before, **:after** — These types are used for the before-daemon and after-daemon methods used by **:daemon** method combination.

:default — If there are no untyped methods among any of the flavors being combined, then the **:default** methods (if any) are treated as if they were untyped. If there are any untyped methods, the **:default** methods are ignored.

Typically, a base flavor defines default methods for certain of the operations understood by its family. When you are using the daemon method combination, which is the default, these default methods are not called if another flavor provides its own method. But with certain strange forms of method combination (**:or**, for example), the base flavor uses a **:default** method to achieve its desired effect.

:or, **:and** — These types are used for **:daemon-with-or**, **:daemon-with-and**, **:or**, and **:and** method combinations. The **:or** methods are wrapped in an **or**, or the **:and** methods are wrapped in an **and**, with the primary method being the last element of the **and** or **or** form, between the **:before** and **:after** methods.

:override — This method type allows the features of **:or** method combination to be used together with daemons. If you specify the method combination type **:daemon-with-override**, you can use **:override** methods. The **:override** methods are executed first until one of them returns true. If this happens, that method's value(s) is returned, and no more methods are called. If all the **:override** methods return **nil**, the **:before**, primary, and **:after** methods are executed as usual.

Typically, the **:override** method usually returns **nil** and does nothing, but in exceptional circumstances, it takes over the handling of the operation.

:case — The **:case** methods are used by **:case** method combination. These method types can be used with any method combination type. They have standard meanings independent of the method combination type being used. For more information, see the description of **:case** method combination earlier in this numbered paragraph.

:around — An **:around** method is able to control when, whether, and how the remaining methods are executed. It is given a continuation, which is a function that executes the remaining methods. This method has complete responsibility for calling the continuation or not and for deciding which arguments to give it. For the simplest behavior, the arguments are the operation name and operation arguments that the **:around** method itself received, but sometimes the whole purpose of the **:around** method is to modify the arguments before the remaining methods see them.

The **:around** method receives three special arguments before the arguments of the operation itself: the *continuation*, the *mapping-table*, and the *original-argument-list*. (For more information about mapping tables, see paragraph 19.13, Implementation of Flavors.) The last is a list of the operation name and operation arguments. The simplest way for the **:around** method to invoke the remaining methods is to execute the following:

(lexpr-funcall-with-mapping-table
continuation mapping-table original-argument-list)

In general, the *continuation* form is called with either **funcall-with-mapping-table**, or **lexpr-funcall-with-mapping-table**, provided that the *continuation* form, the *mapping-table* form, and the operation name (which you know because it is the same as in the **defmethod**), are followed by whatever arguments the remaining methods are supposed to see.

The following form defines a mixin that modifies the `:set-foo` operation so that the value actually used in it is one greater than the value specified in the message:

```
(defflavor foo-one-bigger-mixin () ())
(defmethod (foo-one-bigger-mixin :around :set-foo)
           (cont mt dummy new-foo)
            (declare (ignore dummy))
      (funcall-with-mapping-table cont mt :set-foo (1+ new-foo)))
```

:wrapper — This type of method is used internally by **defwrapper**. Note that if one flavor defines both a wrapper and an **:around** method for the same operation, the **:around** method is executed inside the wrapper.

:combined — This type of method is used internally for automatically generated combined methods.

:inverse-around — Methods of this type work just like **:around** methods but are invoked at a different time and in a different order.

With **:around** methods, those of earlier flavor components are invoked first, starting with the instantiated flavor itself, and those of earlier components are invoked with them. However, **:inverse-around** methods are invoked in the opposite order: **sys:vanilla-flavor** would come first. Also, all **:around** methods and wrappers are invoked inside all the **:inverse-around** methods.

For example, the **:inverse-around :init** method for **w:sheet** (a base flavor for all window flavors) is used to handle the init keywords **:expose-p** and **:activate-p**, which cannot be handled correctly until the window is entirely set up. They are handled in this method because it is guaranteed to be the first method invoked by the **:init** operation on any flavor of window (because no component of **w:sheet** defines an **:inverse-around** method for this operation). All the rest of the work of making a new window valid takes place in the method's continuation; when the continuation returns, the window must be as valid as it will ever be, and it is ready to be exposed or activated.

:progn, :list, :inverse-list — Each of these method types is allowed in the method combination style of the same name. In those method combination styles, these typed methods work just like untyped ones, but all the typed methods are called before any of the untyped ones.

vanilla-flavor

19.8 The operations described in this paragraph are a standard protocol that all message-receiving objects are assumed to understand. The standard methods that implement this protocol are automatically supplied by the flavor system unless you specifically tell it not to do so. These methods are associated with the flavor **sys:vanilla-flavor**:

sys:vanilla-flavor Flavor

Unless you specify otherwise (with the **:no-vanilla-flavor** option to **defflavor**), every flavor includes the **vanilla-flavor**, which has no instance variables but provides some basic useful methods.

:print-self *stream printdepth escape-p* Method of **sys:vanilla-flavor**

This method outputs the printed representation of an instance to a stream. The printer sends this message when it encounters an instance. The arguments specify the stream, the current depth in list structure (for comparison with ***print-level***), and whether printing with escaping is enabled (**prin1** vs **princ**; see the *Explorer Input/Output Reference* manual). The **vanilla-flavor** ignores the last two arguments and prints something such as *#<flavor-name octal-address>*. The *flavor-name* tells you what type of object it is, and the *octal-address* allows you to distinguish different objects. (Note that if the garbage collector is active, the object may be moved around such that the address will change.)

:describe Method of **sys:vanilla-flavor**

This method describes an object instance, printing a description onto the ***standard-output*** stream. The **describe** function sends this message when it encounters an instance. The **vanilla-flavor** outputs (in a reasonable format) the object, the name of its flavor, and the names and values of its instance variables.

:set *keyword value* Method of **sys:vanilla-flavor**

This method sets the internal value specified by *keyword* to the new value specified by *value*. For flavor instances, the **:set** operation uses **:case** method combination, and a method is generated automatically to set each settable instance variable, with *keyword* being the variable's name as a keyword.

:which-operations Method of **sys:vanilla-flavor**

This method returns a list of the operations its flavor instance can handle. The **vanilla-flavor** generates the list once per flavor and remembers it, minimizing consing and computing time. If a new method is added, the list is regenerated the next time someone asks for it.

:operation-handled-p *operation* Method of **sys:vanilla-flavor**

For this method, the argument *operation* is an operation name. This method returns a true value if it has a handler for the specified operation or **nil** if it does not. If there is a **:default** *operation* method, **:operation-handled-p** returns true. If *operation* is handled by virtue of the **:default-handler** **defflavor** option, **:operation-handled-p** returns **nil**.

:get-handler-for *operation* Method of **sys:vanilla-flavor**

For this method, the argument *operation* is an operation name. This method returns the method it uses to handle *operation*. If it has no handler for that operation, it returns **nil**. This method is like the **get-handler-for** function, but, of course, you can use it only on objects known to accept messages.

:send-if-handles *operation* &rest *arguments* Method of **sys:vanilla-flavor**

For this method, the parameter *operation* is an operation name, and *arguments* is a list of arguments for the operation. If the instance handles the operation, it sends itself a message with the specified operation and arguments and returns whatever *operation* returns. If it does not handle the operation, it simply returns **nil**.

The following three operations are implemented in a special way that may cause unwanted or surprising side effects. Before the indicated function or form is evaluated, all of the instance variables for the object are bound specially so that your form can produce its proper side effect. Specifically, if you perform a **setq** of an instance variable, the **setq** modifies the value in the object, However, this also means that all functions called by your form are also able to access these special variables.

:break Method of **sys:vanilla-flavor**

When this method is called, special variables with the names of the instance variables are bound to the values of the instance variables, and the function **break** is called.

:eval-inside-yourself *form* Method of **sys:vanilla-flavor**

For this method, the argument specifies a form that is evaluated in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. Consequently, you can use **setf** on one of these special variables; the instance variable is modified. This method is intended to be used mainly for debugging.

:funcall-inside-yourself *function* &rest *args* Method of **sys:vanilla-flavor**

For this method, the argument *function* is applied to *args* in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. Consequently, you can use **setf** on one of these special variables and modify the instance variable. This method allows callers to provide actions to be performed in an environment set up by the instance.

Property List Operations

19.9 It is often useful to associate a property list with an abstract object for the same reasons that it is useful to have a property list associated with a symbol. This paragraph describes a mixin flavor that can be used as a component of any new flavor in order to provide that new flavor with a property list. Note that this mixin gives each instance its own unique property list. The usual property list functions (**get**, **putprop**, and so forth) all work on instances by sending the instance the corresponding message.

sys:property-list-mixin Flavor

This mixin flavor provides the basic operations on property lists.

:get *property-name* &optional *default* Method of **sys:property-list-mixin**

This method looks up the object's *property-name*. If it finds such a property, it returns the value; otherwise, it returns *default*, which defaults to **nil**.

:getl *property-name-list* Method of **sys:property-list-mixin**

This method is like the **:get** operation, except that the argument is a list of property names. The **:getl** method searches down the property list until it finds a property whose property name is one of the elements of *property-name-list*. It returns the portion of the property list beginning with the first such property that it finds. If it does not find any, it returns **nil**.

:putprop *value property-name* Method of **sys:property-list-mixin**

This method gives the object a *property-name* of *value*.

:remprop *property-name* Method of **sys:property-list-mixin**

This method removes the object's *property-name* property by splicing it out of the property list. The returned value is the same as for the **remprop** function. That is, if a propery is removed, a non-**nil** value is returned; if no such property is found, **nil** is returned.

:get-location *property-name* Method of **sys:property-list-mixin**

This method returns a locative pointer to the cell in which this object's *property-name* is stored. If there is no such property, a cell is added to the property list and initialized to **nil**, and a pointer to that cell is returned. This method never returns **nil**.

:push-property *value property-name* Method of **sys:property-list-mixin**

This method sets the value of the *property-name* of the object to a list whose car is *value* and whose cdr is the former value of the *property-name* of the list. The value of the *property-name* of the object must be a list (note that **nil** is a list, and an absent property is **nil**). This method is analogous to performing the following:

(**push** *value* (**get** *object property-name*))

:plist Method of **sys:property-list-mixin**

This method returns the current value of the property list.

:set-plist *list* Method of **sys:property-list-mixin**

This method sets the property list to the value of *list*.

:property-list-location Method of **sys:property-list-mixin**

This method returns a locative pointer to the cell in the instance that holds the property list data.

Printing Flavor Instances Readably

19.10 A flavor instance can print out so that it can be read back in, as long as you give it a **:print-self** method that produces a suitable printed representation and as long as you provide a way to parse it. The convention for printing an instance readably is to print it as follows:

#⊂ *flavor-name additional-data* ⊃

You must also make sure that the flavor defines or inherits a **:read-instance** method that can parse the *additional-data* and return an instance. A convenient way of doing this is to use **sys:print-readably-mixin**.

sys:print-readably-mixin — Flavor

This mixin allows flavor instances to be printed in Lisp-readable form.

:print-self *stream* — Method of **sys:print-readably-mixin**

This method writes a form to *stream*. When the form is read in, the Reader recreates the instance. Specifically, **:print-self** writes the Reader macro #⊂ followed by the flavor name. Then, it calls **:reconstruction-init-plist**, which returns an alternating list of keywords and values that are passed to **make-instance** to recreate this instance. The **:print-self** method then writes this list and the ⊃ characters to *stream*.

:read-instance *flavor stream* — Method of **sys:print-readably-mixin**

This method is called by the Lisp Reader to read and parse the reconstruction *plist* options written by **:print-self**. This method gives the reconstruction property list an alternating list of keywords and values to be used as an initialization property list to **make-instance**. This method creates the new instance, which is also the returned value.

:reconstruction-init-plist — Default Method of **sys:print-readably-mixin**

This default method covers the simple cases of writing the reconstruction property list. Specifically, it supplies a property list with each of the inittable instance variables and their values. If your flavor has daemon **:init** methods that expect to find initialization keywords, you must supply your own **:reconstruction-init-plist** method to generate the required arguments. Daemon methods around this fault method cannot alter the returned value.

Hash Table Operations

19.11 The Explorer system also supports a hash table flavor that can be mixed in to your flavor definitions or can be used in a standalone fashion. The normal hash table operations work on a hash table flavor object by simply passing the corresponding message. Section 11, Hash Tables, for more information.

hash-table-mixin — Flavor
eq-hash-table-mixin — Flavor
equal-hash-table-mixin — Flavor

Each of these flavors can be instantiated by itself or be mixed into a flavor of your choice to add a hash table capability. The **hash-table-mixin** flavor defaults to an eq hash table.

:size Operation on **hash-table**

This operation returns the number of entries in the hash table. Note that the hash table is rehashed when only a fraction of this number of entries (the rehash threshold) are full.

:filled-entries Operation on **hash-table**

This operation returns the number of entries currently occupied in the hash table.

:get-hash *key* Operation on **hash-table**
:put-hash *key* &rest *values* Operation on **hash-table**
:swap-hash *key* &rest *values* Operation on **hash-table**
:rem-hash *key* Operation on **hash-table**
:map-hash *function* &rest *extra-args* Operation on **hash-table**
:map-hash-return *function* Operation on **hash-table**
:clear-hash Operation on **hash-table**

These operations are equivalent to the functions **get-hash**, **puthash**, **swaphash**, **remhash**, **maphash**, **maphash-return**, and **clrhash**, except that the hash table need not be specified as an argument because it is the object that receives the message. These functions actually work by invoking the corresponding operations.

:describe Operation on **hash-table**

This operation returns the following information about the hash table: the number of entries, the maximum size, whether the hash table is locked, the number of entries deleted, the rehash threshold, the number of values associated with each key, and the function to perform the rehash.

:fasd-form Operation on **hash-table**

This operation returns a form that, when evaluated, reconstructs the hash table instance. This form is usually called by the **dump-forms-to-file** function when writing objects to an object file.

:modify-hash *key function* &rest *additional-args* Operation on **hash-table**

This operation passes to *function* the value associated with *key* in the hash table. Whatever *function* returns is stored in the table as the new value for *key*. Thus, the hash association for *key* is both examined and updated according to *function*.

The arguments passed to *function* are *key*, the value associated with *key*, a flag (**t** if *key* is actually found in the hash table), and any *additional-args* that you specify.

If the hash table stores more than one value per key, then only the first value is examined and updated.

Wrappers and Whoppers

19.12 The following macros supply a specialized control form for a kind of method combination. They are useful in only a small number of cases.

defwrapper (*flavor-name operation*) (*lambda-list . inner-body*) {*wrapper-macro-form*}* Macro

Sometimes the way in which the flavor system combines the methods of different flavors (the daemon system) is not powerful enough. In this case, **defwrapper** can be used to define a macro that expands into code that is wrapped around the invocation of the methods. For example, suppose you need a lock closed during the processing of the `:foo` operation on flavor `bar`, which takes two arguments, and you have a `lock-frobboz` special form that knows how to perform the lock (presumably it generates an **unwind-protect**). The `lock-frobboz` form needs to see the first argument to the operation to find out what sort of operation is going to be performed (read or write):

```
(defwrapper (bar :foo) ((arg1 arg2) . inner-body)
  `(lock-frobboz (self arg1)
    . ,inner-body))
```

The use of the `inner-body` macro argument prevents the `defwrapper` macro from knowing the exact implementation and allows several `defwrappers` from different flavors to be combined properly.

Note that the argument variables `arg1` and `arg2` are not referenced with commas before them. These appear to be **defmacro** argument variables, but they are not. These variables are not bound at the time the `defwrapper`-defined macro is expanded and the backquoting is performed; rather, the result of the macro expansion and backquoting is code that, when a message is sent, binds these variables to the arguments in the message as local variables of the combined method.

Consider another case. Suppose you thought you wanted a **:before** daemon but found that if the argument is **nil**, you need to return from processing the message immediately without executing the primary method. You can write a wrapper such as the following:

```
(defwrapper (bar :foo) ((arg1 arg2) . inner-body)
  `(cond ((null arg1) (ignore))                 ; Do nothing if arg1 is nil
      (t before-code
              . ,inner-body)))
```

Suppose you need a variable for communication among the daemons for a particular operation; perhaps the **:after** daemons need to know what the primary method did, and this information cannot be easily deduced from only the arguments. You can use an instance variable for this procedure, or you can create a special variable that is bound during the processing of the operation and used freely by the methods. For example:

```
(proclaim '(special *communication*))
(defwrapper (bar :foo) (&rest ignore . inner-body)
  `(let ((*communication* nil))
    . ,inner-body))
```

Similarly, you can use a wrapper that puts a **catch** around the processing of an operation so that any one of the methods can throw out in the event of an unexpected situation.

Like daemon methods, wrappers use an outside-in order: when you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component flavors. However, *all* wrappers are processed before *any* daemons are processed. When the combined method is built, the calls to the before-daemon methods, primary methods, and after-daemon methods are all placed together, and then the wrappers are placed around them. Thus, if a component flavor defines a wrapper, methods added by new flavors execute within that wrapper's context.

Furthermore, **:around** methods can perform some of the same operations that wrappers can. If one flavor defines both a wrapper and an **:around** method for the same operation, the **:around** method is executed inside the wrapper.

Be careful about inserting the inner body into an internal lambda expression within the wrapper's code. This code interacts with internal details of the way combined methods are implemented. This insertion can be done if it is done carefully, but using an **:around** method instead is much simpler.

The following three macros are made available for compatibility with other Lisp implementations of the flavor system.

defwhopper (*flavor-name operation*) *lambda-list* {*body*}* Macro

This macro is similar to a wrapper but uses an **:around** daemon. It has the advantage of hiding some of the flavor implementation details of the **:around** daemon. It is implemented by the following:

```
(defmacro defwhopper ((flavor-name operation) arglist &body body)
   `(defmethod (,flavor-name :around ,operation)
        (.continuation. .mapping-table. .around-args. . ,arglist)
         . ,body))
```

The variables `.continuation.`, `.mapping-table.`, and `.around-args.` are local variables set to environment values when the `:around` method is entered. These variables are subsequently used by **continue-whopper** or **lexpr-continue-whopper**. For a precise explanation of these values, see the description of the **:around** daemon in paragraph 19.6, Method Combination Type.

continue-whopper {*args*}* Macro

This macro works in conjunction with **defwhopper**. When, during the processing of a **defwhopper** body, the combined method calling must be resumed, this macro should be called. The *args* are the arguments that should be passed to the combined method in question. This macro is implemented by the following:

```
(defmacro continue-whopper (&rest arguments)
   `(funcall-with-mapping-table .continuation. .mapping-table.
        (car .around-args.)
         . ,arguments))
```

The `.continuation.`, `.mapping-table.`, and `.around-args.` variables are those described in the **defwhopper** macro definition.

lexpr-continue-whopper *{args}** Macro

This macro performs the same operation as **continue-whopper** except that the last argument is a list of arguments to be passed. This macro is implemented by the following:

```
(defmacro lexpr-continue-whopper (&rest arguments)
    `(lexpr-funcall-with-mapping-table .continuation. .mapping-table.
        (car .around-args.)
         . ,arguments))
```

The `.continuation.`, `.mapping-table.`, and `.around-args.` variables are those described in the **defwhopper** macro definition.

Implementation of Flavors

19.13 An object that is an instance of a flavor is implemented using the data type **dtp-instance**. The representation is a structure whose first word, tagged with **dtp-instance-header**, points to a structure (known to the microcode as an *instance descriptor*) containing the internal data for the flavor. The remaining words of the structure are value cells containing the values of the instance variables. The instance descriptor is a structure that appears on the sys:**flavor** property of the flavor name. It contains, among other things, the name of the flavor, the size of an instance, the table of methods for handling operations, and information for accessing the instance variables.

The macro **defflavor** creates such a data structure for each flavor and links them together according to the dependency relationships between flavors.

A message is sent to an instance simply by calling it as a function, with the operation as the first argument. The microcode binds **self** to the object and binds those instance variables that are defined to be special to the value cells in the instance. Then it passes on the operation and arguments to a hash table that can have **funcall** invoked on it and that is taken from the flavor structure for this flavor.

When this hash table is called as a function, it hashes the first argument (the operation) to find a function to handle the operation and an array called a *mapping table*. The variable **sys:self-mapping-table** is bound to the mapping table, which tells the microcode how to access the other instance variables, those not defined to be special or ordered. Then the function is called. If there is only one method to be invoked, this function is that method; otherwise, the method is an automatically generated function called the *combined method*, which calls the appropriate methods in the correct order. If there are wrappers, they are incorporated into this combined method.

The mapping table is an array whose elements correspond to the instance variables accessible by the flavor to which the currently executing method belongs. Each element contains the position in **self** of that instance variable. This position varies with the other instance variables and component flavors of the flavor of **self**.

Each time the combined method calls another method, it sets up the mapping table required by that method, not in general the same one that the combined method itself uses. The mapping tables for the called methods are extracted from the array leader of the mapping table used by the combined method, which is kept in a local variable of the combined method's stack frame while **sys:self-mapping-table** is set to the mapping tables for the component methods.

sys:self-mapping-table Variable

This variable holds the current mapping table, which tells the running flavor method where in **self** to find each instance variable.

Ordered instance variables are referred to directly without going through the mapping table. This procedure is slightly faster and reduces the amount of space needed for mapping tables. This procedure is also the reason why compiled code contains the positions of the ordered instance variables and must be recompiled when they change.

Order of Definition

19.13.1 You have a certain amount of freedom in choosing the order in which you execute **defflavor, defmethod,** and **defwrapper**. This freedom is designed to make it easy to load programs containing complex flavor structures without having to do things in a certain order. Similarly, not all the methods for a flavor need to be defined in the same file. Thus, the partitioning of a program into files can be along modular lines.

Before a method can be defined (with **defmethod** or **defwrapper**), its flavor must have been defined (with **defflavor**). This requirement makes sense because the system has to have a place to remember the method and because it has to know the instance variables of the flavor if the method is to be compiled.

When a flavor is defined (with **defflavor**), all of its component flavors need not be defined already. This feature is to allow **defflavor** forms to be spread between files according to the modularity of a program and to provide for mutually dependent flavors. Methods can be defined for a flavor with some component flavors not yet defined; however, in certain cases, compiling these methods produces a warning that an instance variable was declared special (because the system did not realize it was an instance variable). If this happens, you should define the undefined component flavors and recompile. The methods automatically generated by the **:gettable-instance-variables** and **:settable-instance-variables defflavor** options (paragraph 19.5, **defflavor** Options) are generated at the time the **defflavor** is executed.

The first time a flavor is instantiated, or when **compile-flavor-methods** is executed, the system looks through all of the component flavors and gathers various information. At this point, an error is signaled if all of the components have not been defined. This is also the time at which certain other errors are detected, for instance, lack of a required instance variable (see the **:required-instance-variables defflavor** option in paragraph 19.5, **defflavor** Options). The combined methods are generated at this time also, unless they already exist.

Changing a Flavor

19.13.2 You can change anything about a flavor at any time. You can change the flavor's general attributes by executing another **defflavor** with the same name. You can add or modify methods by executing **defmethod** several times. If you execute a **defmethod** with the same flavor name, operation (and suboperation if any), and (optional) method type as an existing method, that method is replaced by the new definition. You can remove a method entirely with **undefmethod** (see paragraph 19.4, Flavor Functions).

Unless you have requested special compiler optimizations, these changes always propagate to all flavors that depend on the changed flavor. Normally, the system propagates the changes to all existing instances of the changed flavor and all flavors that depend on it. However, this propagation is not possible when the flavor has been changed so drastically that the old instances would not work properly with the new flavor. This problem occurs if you change the number of instance variables, which changes the size of an instance. It also happens if you change the order of the instance variables (and hence the storage layout of an instance) or if you change the component flavors (which can change several subtle aspects of an instance). The system does not keep a list of all the instances of each flavor, so it cannot find the instances and modify them to conform to the new flavor definition. Instead, it gives you a warning message, on the ***error-output*** stream, that the flavor was changed incompatibly, and the old instances do not get the new version.

The system leaves the old flavor data structure intact (the old instances continue to point at it) and makes a new one to contain the new version of the flavor. If a less drastic change is made, the system modifies the original flavor data structure, thus affecting the old instances that point at it. However, if you redefine methods in such a way that they work only for the new version of the flavor, then trying to use these methods with the old instances does not work.

ERROR HANDLING

20

Introduction

20.1 Programs often encounter unusual situations, such as errors, to which they wish to draw attention. Programmers want to control the behavior of their programs when such situations occur. This section describes the facilities provided by the Explorer system to do both. Note that Common Lisp does not yet define a full error-handling system, so much of this section describes features that may not be portable.

There are three stages in dealing with unusual situations. The first stage is to detect the situation and announce it: this is called *signaling*. The object that represents the event is called a *condition* or *condition instance*. In the Explorer system, a condition is a flavor instance whose base flavor is **condition**. Errors are a subclass of conditions and are built on the flavor **error**. (See Section 19, Flavors). Specific conditions are discussed later in this section.) Condition instances hold information about the particular event being signaled.

The second stage is to look for a *handler* for the condition. When a condition is signaled, the system (conceptually) looks through the functions currently in progress for a handler for that condition; handlers have dynamic scope. A handler, if found, deals with the condition. The handler can correct the situation, ignore it, start over, or do almost anything. The condition mechanism is merely a convenient way to find the appropriate handler for a particular situation. If no handler is found and the condition is an error, the debugger is entered; the debugger can be thought of as the ultimate handler.

The third stage in dealing with unusual situations is proceeding or continuing from the situation. *Proceeding* is what handlers do if they choose to correct the situation; the debugger usually offers a set of commands to do the same. (*Proceeding* is the term used most often in Lisp Machine history; *continuing* is the term used in Common Lisp. These terms are interchangeable.) Several Common Lisp functions described in this section have a **c** in front of their names to indicate that the errors they signal are continueable.

An important part of the signaling-handling-proceeding arrangement is the classification of conditions. The signaler has usually detected a particular kind of error for which only certain kinds of handlers are appropriate and only certain ways of proceeding make sense. The condition instance includes information called *condition names* that describe and classify the condition. (Condition names are analogous to types. The type of a condition instance is usually one of its condition names, but the condition-name hierarchy is not the same as the type hierarchy.) Condition names are used to match conditions with handlers. Each handler specifies one or more condition names to which it applies, and if the handler and the condition have at least one condition name in common, the handler applies to the condition.

A given condition instance can (and usually does) have more than one condition name. The condition **sys:divide-by-zero**, for example, also has the condition names **sys:arithmetic-error** and **error**, so it would be handled by handlers for arithmetic errors and errors in general, as well as by handlers for the specific condition. The condition **math:singular-matrix** also has the condition names **sys:arithmetic-error** and **error** because it is an arithmetic error but is still distinguishable from **sys:divide-by-zero**. All conditions also have the condition name **condition**. Not all conditions are errors: the **:fquery** condition, signaled by **y-or-n-p** and other functions, does not have the condition name **error**, so it is not an error and does not cause the debugger to be entered.

Condition names need not be arranged in a strict hierarchy, though they usually are. For example, there could be a condition **remote-disk-full** that has condition names **fs:no-more-room-error** and **sys:remote-network-error**. There are other kinds of remote network errors and there are local disk-full errors, but this condition combines parts of both.

Condition instances contain other information also. This information is usually specific to the kind of condition and is discussed later in paragraph 20.5.2, Basic Condition Operations, and in the descriptions of the conditions themselves. Conditions are described throughout the Explorer manual set. The following description of **sys:divide-by-zero** is provided as an example; some basic conditions are described later in this section.

The **sys:divide-by-zero** condition is typical of other conditions. Its explanation is expanded here as an example of the general properties of conditions.

sys:divide-by-zero (sys:arithmetic-error error) Condition

This condition name is always accompanied by **sys:arithmetic-error** and **error** (that is, it categorizes a subset of these categories). The presence of **error** implies that all **sys:divide-by-zero** conditions are errors.

The condition instance signaled by dividing by zero handles the **:function** operation by returning the function that performed the division (it might be **truncate**, **floor**, **ceiling**, or **round**, as well as **/**). In general, for each condition name there are conventions indicating what additional information is provided and what operations are used to obtain it.

The flavor of the condition instance and its component flavors are always included in the condition names (except that **sys:vanilla-flavor** and some other flavor components are omitted, since they are not useful categories for condition handlers to specify). In the preceding example, the flavor of the condition is **sys:arithmetic-error**, and its components include **error** and **condition**. The symbol **sys:divide-by-zero** is just a condition name, not a flavor. Condition names require new flavors only when they require significantly different handling by the error system.

Signaling Conditions

20.2 The basic functions for signaling conditions are **make-condition** and **signal-condition**, which are described in paragraphs 20.5.4, Creating Condition Instances, and 20.5.5, Signaling a Condition Instance, respectively. Usually, it is easier to use one of several functions that handle the details of a condition in a convenient way. These functions are explained in the following descriptions.

error *format-string* &rest *format-args* *[c]* Function
error *signal-name* &optional *format-string format-args* Function
ferror *signal-name* &optional *format-string* &rest *format-args* Function
ferror *format-string* &rest *format-args* Function

These functions are generally used to signal fatal errors—those without any way to proceed. The format string (and its associated arguments) define a message indicating what type of error occurred. This is often all you need to signal an error.

Note that the first argument can be either a *format-string* or a *signal-name*. The **error** and **ferror** functions try to be compatible with Common Lisp, Zetalisp, and MacLisp, so they accept two argument patterns. If the first argument is a string, it is taken to be the *format-string* and *signal-name* is assumed to be **nil**. Otherwise, the first argument is *signal-name*.

The *signal-name* argument can be a condition instance to be signaled with **signal-condition**, a signal name defined by **defsignal**, the condition flavor name to be signaled, a condition name to be included in the signal, or **nil** (the signal name for a general fatal error).

Both *format-string* and *format-args* are usually additional arguments passed to **format**, but they can be overridden by the definition of the signal name. For example:

```
(ferror 'sys:negative-sqrt
           "You cannot take the square root of ~S. " number)
```

Arguments not used by the *format-string* are not used for printing the error message, but the signal name may still expect them to be present as part of its definition.

The **ferror** function is compatible with the Common Lisp **error** function if the first argument to **ferror** is **nil** or a *format-string*.

cerror *continue-format-string error-format-string* &rest *format-args* *[c]* Function
cerror *proceed-type ignore* &optional *signal-name format-string* &rest *format-args* Function

This function signals an error and provides a single way to proceed from that error. The *continue-format-string* argument (along with any *format-args*) is a format string that describes the effect of proceeding after entering the debugger. The *error-format-string* argument (along with any *format-args*) is a format string that is used to produce a brief description of the error. Both the *continue-format-string* and the *error-format-string* use the same *format-args*.

For example, the following is one way that **cerror** can be used to signal an error when a function is given too many arguments.

```
(let ((number-of-arguments-supplied (list-length arglist)))
   (when (> number-of-arguments-supplied maximum-number-of-arguments)
      (cerror "Tries again, dropping the excess arguments."
              "Function ~S was called with too many arguments (~D)."
              function
              number-of-arguments-supplied)
      (setq arglist (subseq arglist 0 maximum-number-of-arguments))))
```

If the debugger is entered, the error message is taken from the *error-format-string* and the *format-args*. The RESUME key is bound to a command that returns from the call to **cerror**, and its documentation is taken from the *continue-format-string* and the *format-args*.

The code following the **cerror** is responsible for correcting the problem. A common idiom is a loop that encloses an error check, a call to **cerror**, and code that prompts for corrected values so that the program does not continue past the error check until it is given correct data. This idiom is embodied in, for example, the macros **assert**, **check-type**, and **check-arg**, which are described later in this section.

The second argument pattern for **cerror** is from Zetalisp (like **error** and **ferror**, **cerror** decides which form to use by determining if the first argument is a string). Paragraphs 20.4, Proceeding, and 20.5, Condition Instances, explain the terms used here. The function first creates a condition instance by passing the *signal-name*, *format-string*, and *format-args* arguments to **make-condition**. It then signals the condition instance with **signal-condition**.

If *proceed-type* is non-**nil**, then it is passed to **signal-condition** as a proceed type. For compatibility with old uses of **cerror**, if *proceed-type* is **t**, **:new-value** is used as the proceed type. If *proceed-type* is **:yes**, **:no-action** is used as the proceed type. If *proceed-type* is **nil**, (specifying no proceed type) **cerror** is identical to **ferror**. The *proceed-type* can also be a list of proceed types.

The second argument to **cerror** is not used and is present only for historical compatibility.

If a user (through the debugger) or a condition handler decides to proceed with a proceed type, the second value that **signal-condition** returns becomes the value of **cerror**. For example:

```
(loop until (typep x 'fixnum)
  do (setf x (cerror :argument-value nil 'sys:wrong-type-argument
             "The argument ~A is ~S, which is not a fixnum." 'x x)))
```

If `x` is not a fixnum, but is **nil**, the condition **sys:wrong-type-argument** is signaled. This provides the proceed type `:argument-value`. If the condition is not otherwise handled, the debugger is entered, and it prints a message similar to the following:

```
The argument x is nil, which is not a fixnum.
```

Then the debugger lists a message similar to the following as one of its proceed types:

```
RESUME        Ask for a replacement argument and proceed.
```

If this proceed type is selected while in the debugger, **cerror** returns the replacement argument. In the preceding example, `x` is set to the value returned from `cerror`.

assert *test-form* [({*place*}*) [*format-string* {*format-arg*}*]] *[c]* Macro

This macro signals an error if *test-form* evaluates to **nil**. Proceeding from this error allows you to alter the values of the variables listed in the *places*; then **assert** reevaluates *test-form*.

Each *place* is a generalized variable that is normally used in *test-form*. You should be able to change the value of *place* for proceeding, which means that you must be able to set *place* with **setf**. Each *place* becomes a proceed type with means for replacing the value of that *place*.

The *format-string* and *format-arg*s arguments are passed to **format** to make the error message. Neither of these arguments is evaluated unless an error is signaled. They are reevaluated if the error is resignaled. If no *format-string* is provided, a generic message appears announcing that the assertion has failed.

When an error occurs, **assert** signals the error **eh:failed-assertion**, providing a proceed type for each *place* that asks for a replacement value.

The following example provides a proceed type with which to replace the value of `x`:

```
(assert (and (numberp x) (plusp x))
        (x)
        "~A is not a positive number" x)
```

The **assert** function returns **nil**.

check-type *place type-spec* &optional *description* *[c]* Macro

This macro signals a correctable error if the value of *place* does not match the type of *type-spec*. The *place* argument has the same restrictions as the *place* argument to **setf** (see Section 2, Symbols). The *type-spec* argument is a type specifier, is a suitable second argument to **typep** (see Section 12, Type Specifiers), and is not evaluated. The following is a simple example:

```
(check-type foo (integer 0 10))
```

This example signals an error unless `foo`'s value is an integer between `0` and `10`, inclusive. The **typep** function is used for the test.

If an error is signaled, the error message contains both the name of the variable or place where the erroneous value was found and the erroneous value itself. An English description of the type of object that was wanted is computed automatically from the type specifier for use in the error message. For the commonly used type specifiers, this computed description is adequate. If it is unsatisfactory in a particular case, you can specify *description*, which is used instead. To make the error message grammatical, *description* should start with an indefinite article.

The error signaled is of condition **sys:wrong-type-argument**. The proceed type **:argument-value** is provided. If a handler proceeds using this proceed type, it should specify one additional argument, which is stored into *place* using **setf**. The new value is then tested, and so on. The **check-type** macro returns when a value passes the test.

check-arg *var-name predicate type-description* &optional *type-symbol* Macro

This macro is useful for checking arguments to make sure that they are valid when a simple type check is not sufficient. It signals an error if the value of *var-name* does not satisfy *predicate*.

The *var-name* argument is the name of the variable to check. If *var-name* does not satisfy *predicate* and the error is proceeded, *var-name* is set to a replacement value. The *predicate* argument is a test for whether the variable is valid. It can be either a symbol whose function definition takes one argument and returns non-**nil** if the argument is correct, or it can be a non-atomic form that is evaluated to check the argument and that presumably contains a reference to the *var-name* variable. The *type-description* argument is a string that expresses *predicate* in English, to be used in error messages. The *type-symbol* argument is a symbol used by condition handlers to determine which type of argument was expected. It can be omitted if it is to be the same as *predicate*, which must be a symbol in this case. For example, if *type-symbol* is **numberp**, a condition handler can tell that a number is needed and might try to convert the actual supplied value to a number and proceed.

The **check-arg** macro actually calls **cerror** with proceed type **:new-value** and signal name **sys:wrong-type-argument**.

The following is a simple example of this macro:

```
(check-arg sym (or (symbolp sym)
                   (and (listp sym)
                        (every ´symbolp sym)))
               "a symbol or a list of symbols")
```

The value of the argument `sym` should be a symbol or a list of symbols. If `sym` is 3 and the debugger is entered, the debugger prints a message similar to the following:

```
>>Error: The argument SYM was 3, which is not a symbol or a list of
         symbols.
```

Sometimes it is necessary to encode the predicate for a handler that is to examine the condition signaled by **check-arg**. For these cases, supply a unique symbol as the *type-symbol* argument. For example:

```
(check-arg sym (or (symbolp sym)
                   (and (listp sym)
                        (every ´symbolp sym)))
           "a symbol or a list of symbols"
           symbol-or-list-of-symbols)
```

The effects are the same if the debugger is entered, but handlers can use the **:description** message to distinguish between different errors.

The **check-arg** macro uses *predicate* to determine whether the value of the variable is of the correct type. If it is not, **check-arg** signals the **sys:wrong-type-argument** condition. If a handler proceeds, using proceed type **:new-value**, the variable is set to the value proceeded with, and **check-arg** starts over, checking the type again.

In general, what constitutes a valid argument is specified in three ways in **check-arg**. The *type-description* argument is human-understandable, *type-symbol* is program-understandable, and *predicate* is executable. You must ensure that these three specifications agree.

Aside from the *type-symbol* argument, `(check-arg v p d)` is equivalent to `(check-type v (satisfies p) d)`.

etypecase *object* {(*type-specifier* {*form*}*)}* *[c]* Macro

The name of this macro stands for error-checking **typecase**. It executes the *forms* of the first clause whose *type-specifier* matches the data type of *object*'s value. The *type-specifiers* are unevaluated. (The first element of each clause is a type specifier, used as the second argument to **typep** to test the type of *object*.)

The values of the last form are the values of **etypecase**. If no clause matches, an uncorrectable error is signaled, using **error**. The **etypecase** macro is similar to **typecase**, but it has no **otherwise** or **t** clause.

ctypecase *object* {(*type-specifier* {*form*}*)}* *[c]* Macro

The name of this macro stands for continuable error-checking **typecase**. It executes the *forms* of the first clause whose *type-specifier* matches the data type of *object*'s value. The *type-specifiers* are unevaluated. (The first element of each clause is a type-specifier, used as the second argument to **typep** to test the type of *object*.) The values of the last form are the values of **ctypecase**.

If no clause matches, a correctable error (using **cerror** with condition **sys:wrong-type-argument**) is signaled. If you proceed from this error, a new value that you specify replaces the old value of *object* (which must meet the requirements of the *place* argument to **setf**), and the **ctypecase** is tried again. The **ctypecase** macro is similar to **typecase** in construction, but it has no **otherwise** or **t** clause.

ecase *test-object* {(*test-form* {*form*}*)}* *[c]* Macro

The name of this macro stands for error-checking **case**. It executes the *forms* of the first clause whose *test-form* matches *test-object*'s value. The value of *test-form* can be a symbol, character, or number match values, which are not evaluated. The *test-object* argument is compared with the match values using **eql**. When a match value matches, the clause's forms are executed, and the value of the last form in the clause is the value of **ecase**.

If no clause matches, an uncorrectable error is signaled via **error**. The **ecase** macro is similar to **case**, but it has no **otherwise** or **t** clause. For example:

```
(ecase x
    ((apples oranges) (foo))
    (nuts (bar)))
```

If the value of `x` is `'bolts`, an error is signaled, and an error message similar to the following is printed:

```
>>Error: The value of x, BOLTS, is not APPLES, ORANGES or NUTS.
```

ccase *test-object* {(*test-form* {*form*}*)}* *[c]* Macro

The name of this macro stands for correctable error-checking **case**. It executes the *forms* of the first clause whose *test-form* matches *test-object*. The value of *test-form* can be a symbol, character, or number. The *test-form* clause is a match value or a list of match values. The *test-object* argument is compared with the match values using **eql**. When a match is found, the clause's forms are executed, and the value of the last form in the clause is the value of **ccase**.

If no clause matches, a correctable error (using **cerror** with condition **sys:wrong-type-argument**) is signaled. You can proceed by giving a new value for *test-object* (which must meet the requirements for the *place* argument to **setf**). The **ccase** macro is similar in construction to **case**, but it has no **otherwise** or **t** clause.

warn *format-string* &rest *format-args* *[c]* Function
break-on-warnings *[c]* Function

This function uses *format-string* and *format-args* to print a message on the ***error-output*** stream and then returns.

If ***break-on-warnings*** is non-**nil**, it calls **break** instead.

error-output *[c]* Variable

This variable indicates the stream for error-message output. It is usually the same as ***standard-output***.

fsignal *format-string* &rest *format-args* Function

This function is used for signaling without specifying a particular signal name. It is equivalent to the following:

`(cerror :no-action nil nil` *format-string format-args*`...)`

signal *signal-name* &rest *make-condition-arguments* Function

This function signals a condition, allowing handlers to proceed with the specified proceed types. Both *signal-name* and *make-condition-arguments* are passed to **make-condition**, and the result is signaled with **signal-condition** (for descriptions of **make-condition** and **signal-condition**, see paragraphs 20.4, Creating Condition Instances, and 20.5.5, Signaling a Condition Instance, respectively). If *signal-name* is **nil**, the condition **ferror** is signaled with proceed types **:new-value** and **:no-action**.

If *make-condition-arguments* are keyword arguments and **:proceed-types** is one of the keywords, the associated value is used as the list of proceed types. In particular, if *signal-name* is actually a condition instance so that the remaining arguments are ignored by **make-condition**, you can specify the proceed types this way.

If no proceed types are specified, **signal-condition** uses a default list of all the proceed types (known to the condition instance) that prompt the user about how to proceed. If a condition handler or the debugger decides to proceed with one of the proceed types, **signal** returns the values of **signal-condition**. The proceed type is always the first value returned.

For example:

```
(signal 'file-error :proceed-types '(:retry-file-operation))
```

Handling Conditions

20.3 Conditions can be handled with varying degrees of discrimination. Several functions can optionally return the condition object rather than signaling the condition. In this case, all errors are simply caught; the returned value can be examined to determine whether an error occurred or the function completed normally. Several macros can be wrapped around arbitrary expressions to catch errors in the same way. Other macros allow different actions to be taken, depending on the type of error or conditional expressions involving the condition object. Finally, other macros set up general handlers that can take any action, or no action, for any condition or conditions. In fact, all the other forms are written in terms of these general-handler macros. The following descriptions explain all the techniques the Explorer system provides for handling conditions.

errorp *object* Function

This function returns true if *object* is a condition instance and one of its component flavors is **error**. This function is equivalent to the following, but it is more efficient and somewhat faster:

```
(typep object 'error)
```

or

```
(condition-typep object 'error)
```

Some functions optionally return the condition instance rather than signaling it if an error occurs. An example of this happens when the function **open** is called with the optional keyword **:error** set to **nil**. The **errorp** function is useful in testing the value returned from such functions.

condition-typep *condition-instance condition-name* Function

This function returns true if *condition-instance* possesses the *condition-name*.

The *condition-name* argument can also be a combination of condition names using **and**, **or**, and **not**. In this case, the condition tested for is a Boolean combination of the presence or absence of various condition names. Consider the following example, where `condition-object` is a condition instance:

```
(condition-typep condition-object 'fs:file-not-found)

(condition-typep condition-object
           '(or fs:file-not-found
                fs:directory-not-found))
```

This function is distinct from **typep** because condition names are not always types.

Simple Condition Handlers

20.3.1 Simple handlers that catch errors are established using the **ignore-errors**, **errset**, and **catch-error** macros.

ignore-errors &body {*body-form*}* Macro

This macro establishes a handler to evaluate the *body-forms* and return, even if an error occurs. If an error occurs, the first value returned is **nil**, and the second is non-**nil**. If there is no error inside *body-forms*, the first value returned is the first value of the last *body-form*, and the second value is **nil**.

catch-error *form* &optional *print-flag* Macro

This macro establishes a handler to evaluate *form* and return even if an error occurs. If an error occurs, the usual error message is printed, unless *print-flag* is **nil**. Two values are returned: the first value is **nil**, and the second is **t**, indicating the occurrence of an error. The *print-flag* argument is evaluated first and is optional, defaulting to **t**.

If no error occurs, the value(s) returned are the value(s) of *form*. Note that this situation creates a possible ambiguity if *form* returns the two values **nil** and **t**. Unfortunately, this macro was designed before multiple values existed.

errset *form* &optional *print-flag* Macro

This macro establishes a handler to evaluate *form* and return, even if an error occurs. If an error occurs, **nil** is returned after the usual error message is printed, unless *print-flag* is **nil**. The *print-flag* argument is evaluated first and is optional, defaulting to **t**. If no error occurs, the value of **errset** is a list of one element—the first value of *form*.

This macro is an old MacLisp form, and its use is not encouraged.

errset Variable

If this variable is non-**nil**, **errset** and **catch-error** are not allowed to trap errors. The debugger may be entered exactly as if there were no **errset**. This arrangement is intended mainly for debugging. The initial value of **errset** is **nil**.

sys:eval-abort-trivial-errors *top-level-form* Function

This function establishes a handler to evaluate *top-level-form* and return its values if no error occurs. On trivial errors (such as **sys:wrong-type-argument**, **too-few-arguments**, **invalid-function-spec**, **unclaimed-message**, or **cell-contents-error**) if the erring argument, variable, or operation appears in a *top-level-form*, the handler asks the user whether to enter the debugger, using **y-or-n-p**.

If the user types N, the handler signals the **sys:abort** condition, which returns to the innermost command loop. If the user types Y, the handler does not handle the condition, but instead allows the debugger to be entered.

More Complex Condition Handlers

20.3.2 Condition handlers that simply throw to the function that established them are very common. The **condition-case** and **condition-call** macros are provided for defining them.

condition-case ({*variable*}*) *body-form* {(*condition-names* {*form*}*)}* Macro

The *body-form* argument is executed with a condition handler established that throws back to the **condition-case** if any of the specified condition names is signaled.

Each list starting with condition names is a *clause* and specifies what to do if one of these condition names is signaled. In the clauses, *condition-names* is either a condition name or a list of condition names; it is not evaluated.

Once the handler has performed the throw, the clauses are tested in order until one is found that applies. This procedure is almost like a **case**, except that the signaled condition can have several condition names, so the first clause that matches any of them is allowed to run. The forms in the clause are executed with the first *variable* bound to the condition instance that was signaled. The other *variable*s are unbound unless there is a **:no-error** clause, in which case they are **nil**. The values of the last form in the clause are returned from **condition-case**.

If none of the specified conditions is signaled during the execution of *body-form* and if no errors are signaled (or if other handlers, established within *body-form*, handle them), then the values of *body-form* are returned from **condition-case**. If a condition not matching any clause is signaled, **condition-case** does not handle it and the debugger may be entered if it is an error.

The *variable* argument can be omitted if it is not used, as in the following example:

```
(condition-case ()
   (print foo)
 (error (format t " <<Error in printing>>")))
```

You can also have a clause starting with **:no-error** in place of a condition name. This clause is executed if *body-form* finishes normally. During the execution of the **:no-error** clause, the variables are bound to the values returned by *body-form*. The values of the last form in the **:no-error** clause are returned from **condition-case**.

condition-call ({*variable*}*) *body-form* {(*condition-predicate-form* {*form*}*)}* Macro

This macro is an extension of **condition-case** that allows you to give each clause an arbitrary conditional expression instead of a simple list of condition names.

The difference between this and **condition-case** is the *condition-predicate-form* in each clause. The clauses in a **condition-call** resemble the clauses of a **cond** rather than those of a **case**.

When a condition is signaled, each *condition-predicate-form* is executed while still within the environment of the signaling (that is, within the actual handler function defined in the macro **condition-call**). The *condition-predicate-form* can refer to the first *variable* to see the condition instance. If any *condition-predicate-form* returns non-**nil**, then the handler throws to the **condition-call**, and the corresponding clause's *form*s are executed. If every *condition-predicate-form* returns **nil**, the condition is not handled by **condition-call**.

In fact, each *condition-predicate-form* is computed a second time after the throw has occurred in order to decide which clause to execute. The code for the *condition-predicate-form* is copied in two different places: once into the handler function to decide whether or not to throw and once in a **cond** that follows the catch.

The *variable*s can be omitted if they are not used; but it is likely that you will need to use the first one.

You can also have a clause starting with **:no-error** in place of a condition name. This clause is executed if *body-form* finishes normally. During the execution of the **:no-error** clause, the variables are bound to the values returned by *body-form*. The values of the last form in the **:no-error** clause are returned from **condition-call**.

Only the first of the *variable*s is used if there is no **:no-error** clause; the others are unbound. Consider the following example:

```
(condition-call (instance)
    (do-it)
  ((condition-typep instance
     '(and fs:file-error (not fs:no-more-room)))
   (compute-what-to-return)))
```

The `fs:no-more-room` condition name is a subcategory of `fs:file-error`. This example handles all file errors *except* for `fs:no-more-room`.

condition-case-if *predicate-form* ({*variable*}*) *body-form* {(*condition-name* {*form*}*)}* Macro

This macro begins by executing *predicate-form*. If it returns non-**nil**, then everything proceeds as for a regular **condition-case**. If *predicate-form* returns **nil**, then the *body-form* is still executed but without establishing the condition handler. The *body-form*'s values are returned, or, if there is a **:no-error** clause, it is executed and its values returned.

condition-call-if *predicate-form* ({*variable*}*) *body-form* {(*condition-predicate-form* {*form*}*)}* Macro

This macro begins by executing *predicate-form*. If it returns non-**nil**, then everything proceeds as for a regular **condition-call**. If *predicate-form* returns **nil**, then the *body-form* is still executed, but without establishing the condition handler. In this case, *body-form*'s values are always returned.

General Condition Handlers

20.3.3 A condition handler is a function that is associated with certain condition names (categories of conditions). The **eh:*condition-handlers*** variable contains a list of the handlers that are current.

eh:*condition-handlers* Variable

This variable is the list of established condition handlers. Each element has the following format:

```
(condition-names function additional-arg-values...)
```

In this example, `condition-names` is a condition name or a list of condition names, or **nil**, which means all conditions, and `function` is the actual handler function.

The `additional-arg-values` are additional arguments to be passed to the `function` when it is called. The `function`'s first argument is always the condition instance; the second argument is the first *additional-arg*, and so on.

eh:*condition-default-handlers* Variable

This variable is the list of established default condition handlers. The format is the same as that of **eh:*condition-handlers***.

When a condition is signaled in a program, the condition-handler list in **eh:*condition-handlers*** is scanned by the system, and all the handlers that apply are called, one by one, until one of the handlers either throws or returns a non-**nil** value. Handlers are established using macros that bind this variable.

Because each new handler is pushed onto the front of **eh:*condition-handlers***, the innermost-established handler gets the first chance to handle the condition. When the handler is run, **eh:*condition-handlers*** is bound so that the running handler (and all those that were established farther in) are not in effect. This arrangement avoids the danger of infinite recursion because of an error in a handler invoking the same handler.

One thing a handler can do is throw to a tag. Often the **catch** for this tag is next to the place where the handler is established, but this does not have to be so.

The handler can also ask to proceed from the condition. It does so by returning a non-**nil** value. For more information, see paragraph 20.4, Proceeding.

The handler can also decline to handle the condition by returning **nil**. Then the next applicable handler is called, and so on, until either a handler does handle the condition or there are no more handlers.

The handler function is called in the environment where the condition was signaled and in the same stack group. All special variables have the values they had at the place where the signaling was performed, and all catch tags that were available at the point of signaling can be thrown to.

Some handlers, such as those defined with **condition-bind**, receive the condition instance as their first argument. When establishing the handler, you can also provide additional arguments to pass to the handler when it is called. This feature allows the same function to be used in varying circumstances.

A second list of handlers is called **eh:*condition-default-handlers***. This list is scanned after all of **eh:*condition-handlers*** has been exhausted; this is the only difference between the two lists. The handlers work in the same way. Default handlers allow some condition handling to occur without interfering with other handlers that may be established outside the default handler.

The fundamental means of establishing a condition handler is with **condition-bind** or **condition-bind-default**.

condition-bind ({(*conditions handler-form* {*additional-arg-form*}*)}*) {*body-form*}* — Macro

This macro executes *body-forms* with one or more condition handlers established.

Each list of *conditions* and *handler-form* establishes one handler. Each *conditions* argument is a condition name or a list of condition names to which the handler should apply. It is *not* evaluated. Each *handler-form* argument is evaluated to produce the function that is the actual handler. The *additional-arg-forms* are evaluated, on entry to the **condition-bind**, to produce additional arguments that are passed to the handler function when it is called. The arguments to the handler function are the condition instance being signaled, followed by the values of any *additional-arg-forms*.

The *conditions* argument can be **nil**; then the handler applies to all conditions that are signaled. In this case, the handler function decides whether to do anything. It is important for the handler to refrain from handling certain conditions that are used for debugging, such as **break** and **sys:call-trap**. The **:debugging-condition-p** operation on condition instances returns non-**nil** for these conditions. Certain other conditions such as **sys:virtual-memory-overflow** should be handled with great care. The **:dangerous-condition-p** operation returns non-**nil** for these conditions. For more information on these conditions, see the paragraph 20.5.2, Basic Condition Operations.

The following example shows how **condition-bind** is used:

```
(defun my-handler (condition-object value-instead)
  (unless (or (send condition-object :dangerous-condition-p)
              (send condition-object :debugging-condition-p))
     (throw 'here value-instead)))

(defun foo (x)
   (condition-bind ((nil 'my-handler 0))
      (catch 'here
         (if (plusp x) (throw 'here x)))))

(foo 3) => 3
(foo -1) => nil
(foo 'a) => 0
```

In the example, `my-handler` declines to handle all debugging conditions and dangerous errors. For all other conditions, it throws to `here` with the value of the *additional-arg-form*, which in this case is `0`.

condition-bind-default ({(*conditions handler-form* {*additional-arg-form*}*)}*) {*body-form*}* — Macro

This macro is like **condition-bind** but establishes a *default handler* instead of an ordinary handler. Default handlers work like ordinary handlers, but they are tried in a different order: first, all the applicable ordinary handlers are given a chance to handle the condition, and then the default handlers get their chance. A more flexible procedure is described under **signal-condition** in paragraph 20.5, Condition Instances.

condition-bind-if *predicate-form* ({(*conditions handler-form* {*additional-arg-form*}*)}*) {*body-form*}* — Macro

This macro begins by executing *predicate-form*. If it returns non-**nil**, then everything proceeds as for a regular **condition-bind**. If *predicate-form* returns **nil**, then the *body-form* is still executed but without establishing the condition handler(s).

condition-bind-default-if *predicate-form* ({(*conditions handler-form* {*additional-arg-form*}*)}*) {*body-form*}* — Macro

This macro is used exactly like **condition-bind-if** but establishes a default handler instead of an ordinary handler.

Proceeding

20.4 Both you (through the debugger) and condition handlers have the option of choosing from among one or more ways to continue execution. Each condition can define, as a convention, certain *proceed types*, which are keywords that signify a certain conceptual way to proceed. For example, the **sys:wrong-type-argument** condition defines the **:argument-value** proceed type that asks for a new value to use as the argument.

When a signaler signals a condition, it can specify proceed types when calling **cerror** and **signal**. Each signaler may or may not implement all the proceed types that are meaningful in general for the condition names being signaled. For example, it is futile to proceed from a **sys:wrong-type-argument** error with **:argument-value** unless the signaler knows how to take the associated value and store it into the argument or to do something else that fits the conceptual specifications of **:argument-value**. For some signalers, this procedure may not make sense at all. Therefore, one of the arguments to **signal-condition** is a list of the proceed types that this particular signaler knows how to handle.

In addition to the proceed types specified by the individual signaler, other proceed types can be provided nonlocally; they are implemented by a *resume handler* that is in effect through a dynamic scope. See the paragraph 20.4.4, Nonlocal Proceed Types.

Proceeding and Handlers

20.4.1 A condition handler can use the **:proceed-types** and **:proceed-type-p** methods on the condition instance to find out which proceed types are available. It can request to proceed by returning one of the available proceed types as a value. This value is returned from **signal-condition**, and the condition's signaler can take action as appropriate.

:proceed-types Method of **condition**

This method returns a list of the proceed types available for this condition instance. The **:proceed-types** method should be used only within the signaling of the condition instance, since it refers to the special variable in which **signal-condition** stores its second argument.

:proceed-type-p *proceed-type* Method of **condition**

This method returns true if *proceed-type* is one of the proceed types available for this condition instance. This method should be used only within the signaling of the condition instance since it refers to the special variable in which **signal-condition** stores its second argument.

If the handler returns more than one value, the remaining values are considered *arguments* of the proceed type. The meaning of the arguments to proceed type and the kind of arguments expected are part of the conventions associated with the condition name that gives the proceed type its meaning. For example, the **:argument-value** proceed type for **sys:wrong-type-argument** errors conventionally takes one argument, which is the new value to use. All the values returned by the handler are returned by **signal-condition** to the signaler.

In the following example, a condition handler proceeds from **sys:wrong-type-argument** errors. This condition handler makes any atom effectively equivalent to **nil** when used in **car** or any other function that expects a list. The handler uses the **:description** operation, which, on **sys:wrong-type-argument** condition instances, returns a symbol describing the data type desired (note that the symbol is not necessarily a type name—it is specific to the error):

```
(condition-bind
   ((sys:wrong-type-argument
          #'(lambda (condition)
                (if (eq (send condition :description) 'cons)
                    (values :argument-value nil)))))
 (setq z (car a)))
```

In this example, the argument to the `:argument-value` proceed type is `nil`. If `a` is 2 (an error), `z` is set to **nil**. If `a` is the list (1 2), `z` is set to 1.

Proceeding and the Debugger

20.4.2 If a condition invokes the debugger, the user can proceed by using a proceed type. When the debugger is entered, each of the available proceed types is assigned a command character starting with SUPER-A. Each character becomes a command to proceed using the corresponding proceed type.

Three additional facilities are needed to make it convenient for the user to proceed using the debugger. Each is provided by methods defined on condition flavors. When you define a new condition flavor, you must provide methods to implement these facilities:

- Documentation — The user must know what each proceed type is for.
- Prompting for arguments — After selecting a proceed type, the user must be prompted for the arguments for the proceed type. Each proceed type can have different arguments to ask for.
- Supplying proceed types — Usually the user can choose among the same set of proceed types that a handler can, but sometimes it is useful to provide the user with a few extra ones or to suppress some of them.

These three facilities are provided by methods defined on condition flavors. Each proceed type that is provided by signalers should be accompanied by suitable methods. Thus, you must normally define a new flavor if you wish to use a new proceed type.

:document-proceed-type *proceed-type stream* Method of **condition**

This method prints the documentation string for a particular proceed type. For example, when sent to a condition instance describing an unbound-variable error, if the proceed type specified is **:new-value**, the text printed is as follows:

```
Proceed, reading a value to use instead.
```

The debugger uses this operation to print the description of a proceed type after printing the command character.

This method prints on *stream* a description of the purpose of *proceed-type*. The **:document-proceed-type** method uses the **:case** method combination (see Section 19, Flavors) to make it convenient to define the way to document an individual proceed type. The string printed should start with an imperative verb form, capitalized, and end with a period. Consider the following example:

```
(defmethod (name-conflict :case :document-proceed-type :skip)
           (stream ignore)
   (format stream "Return without doing the ~A."
       (send self :operation)))
```

As a last resort, if the condition instance has a **:case** method for **:proceed-asking-user** with *proceed-type* as the suboperation and this method has a documentation string, it is printed. In fact, this is the usual way that a proceed type is documented. If no documentation can be found (as might happen with user-defined proceed types), the proceed-type symbol is printed.

To define your own **:document-proceed-type**, use the same pattern as in the preceding example.

:proceed-asking-user *proceed-type continuation read-object-fn* Method of **condition**

This method should prompt the user for suitable arguments to pass with the proceed type. For example, sending **:proceed-asking-user** to an instance of **sys:unbound-variable** with the **:new-value** argument reads and evaluates one expression after displaying the following prompt:

```
Form to evaluate and use instead:
```

The method for **:proceed-asking-user** embodies the knowledge of how to prompt for and read the additional arguments that accompany *proceed-type*.

The **:case** method combination is used (see Section 19, Flavors), making it possible to define the handling of each proceed type individually in a separate function. The documentation string of the **:case :proceed-asking-user** method for a proceed type is also used as the default for the **:document-proceed-type** on that proceed type.

The method for **:proceed-asking-user** should read values by calling *read-object-fn*, using a calling sequence like that of **prompt-and-read**. (The *read-object-fn* may or may not actually use **prompt-and-read**.) After reading the appropriate number and sort of values to go with the particular proceed type, the method should call *continuation*, passing a proceed type and suitable arguments (presumably based on what the user typed). The proceed type passed to *continuation* need not be the same as the one given to **:proceed-asking-user**; it should be one of the proceed types available for handlers to use.

To define your own **proceed-asking-user** method, use the following pattern:

```
(defmethod (condition-flavor :case :proceed-asking-user proceed-type)
                 (continuation read-object-function)
        body...)
```

The *body* argument should prompt the user for input (using `read-object-function`) and call `continuation` with *proceed-type* and any addition arguments.

The following code shows how **sys:proceed-with-value-mixin** provides for the proceed type **:new-value**:

```
(defmethod (sys:proceed-with-value-mixin
                      :case :proceed-asking-user :new-value)
             (continuation read-object-function)
  "Proceeds, reading a value to use instead."
  (funcall continuation :new-value
     (funcall read-object-function
              :eval-read
              "~&Form whose value to use instead: ")))
```

The following is a more complete example that defines three proceed types for the condition **device-error**, a flavor with one instance variable, **device**. The first two proceed types have **:document-proceed-type** methods because the documentation varies depending on the error, whereas the third proceed type just uses the **:proceed-asking-user** method's documentation string. The third proceed type reads an argument; the first two do not.

```
(defmethod (device-error :case :proceed-asking-user :wait)
                         (continuation read-object-function)
   (declare (ignore read-object-function))

   (funcall continuation :wait))

(defmethod (device-error :case :document-proceed-type :wait)
                         (stream ignore)
 (format stream "Proceeds, waiting for the current ~A I/O to complete."
         device))

(defmethod (device-error :case :proceed-asking-user :bash)
                         (continuation read-object-function)
   (declare (ignore read-object-function))
   ;; Terminate the I/O.
  (funcall continuation :bash))

(defmethod (device-error :case :document-proceed-type :bash)
                         (stream ignore)
  (format stream "Proceeds, terminating current ~A I/O." device))

(defmethod (device-error :case :proceed-asking-user :other)
                         (continuation read-object-function)
   "Proceeds, using a different port."
   (funcall continuation :other (funcall read-object-function :string
                                         "Device to use instead of ~A:"
                                         device)))
```

:user-proceed-types *proceed-types* Method of **condition**

This method is given the list of proceed types actually available and is supposed to return the list of proceed types to offer to the user. By default, this operation returns the argument passed to it. All proceed types are available to the user through handlers.

For example, the **sys:unbound-variable** condition conventionally defines the **:new-value** and **:no-action** proceed types. The first specifies a new value; the second attempts to use the variable's current value and receives another error if the variable is still unbound. These are clean operations for handlers to use. However, it is more convenient for the user to be offered only one choice that uses the variable's new value if it is currently bound but asks for a new value otherwise. To offer one choice, a **:user-proceed-types** method replaces the two proceed types with a single one.

Alternatively, you can offer the user two different proceed types that differ only in how they ask the user for additional information. For handlers, there would be only one proceed type.

Assuming that *proceed-types* is the list of proceed types available for condition handlers to return, **:user-proceed-types** returns the list of proceed types that the debugger should offer to the user.

Only the proceed types that are offered to the user need to be handled by **:document-proceed-type** and **:proceed-asking-user**.

The **condition** flavor itself defines this to return its argument. Other condition flavors can redefine this to filter the argument in an appropriate fashion.

The **:pass-on** method combination is used (see Section 19, Flavors) so that if multiple mixins define methods for **:user-proceed-types,** each method has a chance to add or remove proceed types. The methods should not actually modify the argument, but they should cons a new list in which certain keywords are added or removed according to the other keywords that are present.

Elements should be removed only if they are specifically recognized. That is, the method should make sure that any unfamiliar elements present in the argument are also present in the value. You can arrange to omit certain specific proceed types; however, returning only the intersection with a constant list is not legitimate.

The following code is an example of a nontrivial use of **:user-proceed-types:**

```
(defflavor my-error () (error))

(defmethod (my-error :user-proceed-types) (proceed-types)
 (if (member :foo proceed-types)
     (cons :foo-two-args proceed-types)
      proceed-types))

(defmethod (my-error :case :proceed-asking-user :foo)
     (cont read-object-fn)
      "Proceeds, reading a value to foo with."
     (funcall cont :foo
              (funcall read-object-fn :eval-read
               "Value to foo with: ")))

(defmethod (my-error :case :proceed-asking-user :foo-two-args)
     (cont read-object-fn)
     "Proceeds, reading two values to foo with."
     (funcall cont :foo
              (funcall read-object-fn :eval-read
              "Value to foo with: ")
              (funcall read-object-fn :eval-read
              "Value to foo some more with: ")))
```

In this example, if the signaler provides the `:foo` proceed type, then it is one of the SUPER- commands in the debugger, described for the user as the following:

```
Proceeds, reading a value to foo with.
```

If the user chooses this proceed type in the debugger, the following prompt appears:

```
Value to foo with:
```

The value that the user enters is used as the argument when proceeding. In addition, the user is offered the `:foo-two-args` proceed type, which has its own documentation and which reads two values. But for condition handlers, there is actually only one proceed type, `:foo` (the one specified by the signaler). The `:foo-two-args` proceed type is merely a user-visible alternate to the `:foo` proceed type. The `:user-proceed-types` method offers `:foo-two-args` only if the signaler accepts `:foo`.

How Signalers Provide Proceed Types

20.4.3 Each condition name defines a conceptual meaning for certain proceed types, but this does not mean that all of those proceed types can be used every time the condition is signaled. The signaler must specifically implement the proceed types to make them behave conventionally. For some signalers, proceeding may be difficult to perform or may not even make sense. For example, it is no use having a proceed type **:store-new-value** if the signaler does not have a suitable place to permanently store the argument the handler supplies.

Therefore, each signaler is required to specify only those proceed types it implements. Unless the signaler explicitly specifies proceed types one way or another, no proceed types are allowed (except for nonlocal ones, described in paragraph 20.4.4, Nonlocal Proceed Types).

One way to specify the proceed types allowed is to call **signal-condition** (described in paragraph 20.5, Condition Instances) and pass the list of proceed types as the second argument. The **cerror** and **signal** functions call this function specifying proceed types.

Another, less general but more convenient, way to produce the same result is to use **signal-proceed-case**.

signal-proceed-case (({*var*}*) *signal-name* {*signal-name-args*}*) {(*proceed-type* {*form*}*)}* — Macro

This macro is convenient for signaling a condition and providing proceed types. Each clause specifies a proceed type to provide and contains code to be run if a handler proceeds with that proceed type.

A condition instance is created with **make-condition** (described in paragraph 20.5.4, Creating Condition Instances) using *signal-name* and *signal-name-arguments*. The *signal-name-arguments* are the *format-string* and *format-args* for **make-condition**. This condition instance is signaled by calling **signal-condition** and passing to it a list of the proceed types from all the clauses as the list of allowed proceed types.

The *variables* argument is a list of variables bound to the values returned by **signal-condition**, starting with the second value. The first value is tested against the *proceed-type* from each clause, using a **case**. The clause that matches that *proceed-type* executes *forms*. Consider the following example:

```
(defsignal my-wrong-type-arg
    (eh:wrong-type-argument-error sys:wrong-type-argument)
  (old-value arg-name description)
  "Wrong type argument from my own code.")


(signal-proceed-case
           ((newarg)
            'my-wrong-type-arg
            "The argument ~A was ~S, which is not a cons."
            'foo foo)
  (:argument-value (car newarg)))
```

The `my-wrong-type-arg` signal name creates errors with the `sys:wrong-type-argument` condition name. The `signal-proceed-case` signals such an error and handles the `:argument-value` proceed type. If a handler proceeds using this proceed type, the value is put in `newarg`, and then the `car` of `newarg` is returned from the `signal-proceed-case`.

Nonlocal Proceed Types

20.4.4 When the caller of **signal-condition** specifies proceed types with **cerror** and **signal**, these are called *local proceed types* because they are implemented at the point of signaling. There are also *nonlocal proceed types*, which are in effect for all conditions (with appropriate condition names) signaled during the execution of the body of the establishing special form.

For condition handlers, there is no distinction between local and nonlocal proceed types. They are both included in the list of available proceed types returned by the **:proceed-types** operation (all the local proceed types come first). The condition handler selects one by returning the proceed type and any conventionally associated arguments. The debugger's **:user-proceed-types**, **:document-proceed-type**, and **:proceed-asking-user** operations are used in the same way.

The difference between dealing with local and nonlocal proceed types comes after the handler or the debugger returns to **signal-condition**. If the proceed type is a local one (one of those in the second argument to **signal-condition**), **signal-condition** simply returns. If the proceed type is not there, **signal-condition** looks in **eh:*condition-resume-handlers*** for the resume handler associated with the proceed type and calls it. The arguments to the handler function are the condition instance, any additional arguments specified in the resume handler, and any arguments returned by the condition handler in addition to the proceed type. The handler function is supposed to perform a throw. If it returns to **signal-condition**, an error is signaled.

The most general form for establishing a resume handler is with **condition-resume** or **condition-resume-if**, whereas the most common resume handlers can be defined with the variants of **error-restart**.

condition-resume *handler-form* {*body-form*}* Macro

This macro executes *body-form* with a resume handler in effect for a nonlocal proceed type according to the value of *handler-form*.

The value of the *handler-form* should be a list with at least five elements:

({*condition-name*|({*condition-name*}*)}
 proceed-type
 {t|*predicate-function*}
 (*format-string* {*format-arg*}*)
 handler-function
 {*additional-arg*}*)

The `condition-names` element is a condition name or a list of them. The resume handler applies to these conditions only.

The `proceed-type` element is the proceed type implemented by this resume handler.

The `predicate` element is either **t** or a function that is applied to a condition instance and that determines whether the resume handler is in effect for that condition instance.

The `format-string-and-args` element is a list of a string and additional arguments that can be passed to **format** to print a description of what this proceed type is for.

The `handler-function` element is the function called to do the work of proceeding once this proceed type has been returned by a condition handler or the debugger. Its arguments are the condition instance and the `additional-args`. Consider another example:

```
(condition-resume
    '(fs:file-error
      :retry-open
      t
      ("Proceeds, opening the file again.")
      (lambda (ignore) (throw 'tag nil)))
  (loop
    (catch 'tag (return (open pathname)))))
```

The `:retry-open` proceed type is available for all `fs:file-error` conditions signaled within the call to `open`. This particular example also demonstrates a common idiom for retrying that is embodied in **error-restart** (which is described later).

condition-resume-if *predicate-form handler-form* {*body-form*}* Macro

This macro is the same as **condition-resume,** except that the resume handler is in effect only if *predicate-form*'s value is non-**nil**.

eh:invoke-resume-handler *condition-instance proceed-type* &rest *args* Function

This function invokes the innermost applicable resume handler for *proceed-type*. An applicable resume handler is determined by matching its condition names against those possessed by *condition-instance* and by applying its predicate, if not **t**, to *condition-instance*.

If *proceed-type* is **nil**, the innermost applicable resume handler is invoked regardless of its proceed type. However, in this case, the scan stops if **t** is encountered as an element of **eh:*condition-resume-handlers***.

eh:*condition-resume-handlers* Variable

This variable is the current list of resume handlers for nonlocal proceed types. The **condition-resume** macro works by binding this variable. Elements are usually lists that have the format described above under **condition-resume**. The symbol **t** is also meaningful as an element of this list. It terminates the scan for a resume handler when it is made by **signal-condition** for a condition that was not handled. The symbol **t** is pushed onto the list by break loops and the debugger to shield the evaluation of your type-in from automatic invocation of resume handlers established outside the break loop or the error.

You are allowed to use *anonymous* nonlocal proceed types, which have no conventional meaning and are not specially known to the **:document-proceed-type** and **:proceed-asking-user** operations. The anonymous proceed types need not even be symbols, and in practice they are frequently lists consed at run time (often using **with-stack-list**) to make sure they are all distinct. The default definition of **:proceed-asking-user** handles an anonymous proceed type by simply calling the continuation passed to it, reading no arguments. The default definition of **:document-proceed-type** handles anonymous proceed types by passing to **format** the list *format-string-and-args* found in the resume handler.

Anonymous proceed types are treated like other proceed types except as noted above. Proceed types that are lists are treated somewhat specially. For instance, they are all put at the end of the list returned by the **:proceed-types** operation. Also, the debugger command RESUME, which normally proceeds using the first proceed type on that list, does not operate at all if this proceed type is a list.

Anonymous proceed types are usually created with some variant of **error-restart.**

The **error-restart** forms often specify (**error sys:abort**) as the *condition-names*. The presence of **error** causes these forms to be listed and assigned SUPER- command characters by the debugger for all errors. The presence of **sys:abort** causes the ABORT key to use the condition names. These forms are typically used by any sort of command loop so that aborting within the command loop returns to it and reads another command. The **error-restart-loop** macro is often appropriate for simple command loops. The **catch-error-restart** macro is useful when aborting should terminate execution, rather than retry execution. It is also useful with an explicit conditional to test whether a throw was performed.

Most command loops use some version of **error-restart** to set up a resume handler for **sys:abort** so that it returns to the innermost command loop if no handler handles it (as is usually the case). These resume handlers usually apply to **error** as well as **sys:abort** so that the debugger offers a specific command to return to the command loop. For example, the Lisp Listener's read-eval-print loop is similar to the following:

```
(loop
   (catch-error-restart
      ((sys:abort error) "Return to top level in my listener.")
    (setq - (read-for-top-level))
    (princ (eval -))
    (terpri)))
```

All of these variants of **error-restart** can be written in terms of **condition-resume-if.**

error-restart (*condition-names format-string* {*format-args*}*) {*body-form*}* Macro

This macro executes *body-form* with a resume handler and an anonymous proceed type established for *condition-names*. The *condition-names* argument is either a single condition name, a list of condition names, or **nil,** meaning all conditions; it is not evaluated.

The *format-string* and *format-args* arguments, all of which are evaluated, are used by the **:document-proceed-type** operation to describe the anonymous proceed type.

If the proceed type is used for proceeding, the automatically generated resume handler function throws back to the **error-restart,** and the body is executed again from the beginning. If *body-form* returns, the values of the last form in it are returned from **error-restart**. For instance, the example for **condition-resume** could be written as follows:

```
(error-restart (fs:file-error "Proceed, opening the file again.")
     (open pathname))
```

error-restart-if *predicate-form* (*condition-names format-string* {*format-args*}*) {*body-form*}* Macro

This macro is like **error-restart** except that the resume handler is only in effect if the value of the *predicate-form* is non-**nil**.

error-restart-loop (*condition-names format-string format-args*) {*body-form*}* Macro

This macro is like **error-restart** except that it loops to the beginning of *body-form* even if *body-form* completes normally. It is like enclosing an **error-restart** in a **loop**.

catch-error-restart (*condition-names format-string format-args*) { *body-form*}* Macro

This macro is like **error-restart** except that it never loops back to the beginning. If the anonymous proceed type is used for proceeding, the **catch-error-restart** form returns with **nil** as the first value and a non-**nil** second value. If there is no error, the values of the last *body-form* are returned. This situation permits a potential ambiguity if the *body-form* returns **nil** and **t** because this form was designed before multiple values existed.

catch-error-restart-if *predicate-form* (*condition-names format-string format-args...*) {*body-form*}* Macro

This macro is like **catch-error-restart** except that the resume handler is only in effect if the value of the *predicate-form* is non-**nil**.

catch-error-restart-explicit-if *predicate-form* (*condition-names proceed-type format-string* {*format-arg*}*) {*body-form*}* Macro

This macro is similar to **catch-error-restart** except that it executes *body-form* with a resume handler and an explicit proceed type (*proceed-type*) established for *condition-names* instead of an anonymous proceed type.

Condition Instances

20.5 The following paragraphs discuss standard condition flavors, basic condition operations, condition methods used by the debugger, creating condition instances, and signaling a condition instance.

Standard Condition Flavors

20.5.1 The following paragraph contains definitions of some of the standard condition flavors. Other conditions are documented throughout the Explorer manual set.

condition Flavor

This is the base flavor of all conditions and provides a default definition of all the operations described in this section. The **condition** flavor includes the mixin **sys:property-list-mixin**, which defines operations **:get** and **:plist**. Each property name on the property list is also an operation name, so `(send instance :foo)` is equivalent to the following:

```
(send instance :get :foo)
```

The **condition** flavor also provides two instance variables: **eh:format-string** and **eh:format-args**. The method for the **:report** operation on **condition** passes these to **format** to print the error message.

error Flavor

This flavor makes a condition an error condition. The **errorp** function returns true for such conditions, and the debugger is entered if they are signaled and not otherwise handled. Its error message prefix is `>>Error` or `>>Trap`, which indicates that the error was signaled in a program or microcode, respectively.

sys:no-action-mixin Flavor

This mixin provides a definition of the **:no-action** proceed type, which does nothing; it simply proceeds.

sys:proceed-with-value-mixin Flavor

This mixin provides a definition of the **:new-value** proceed type, which proceeds, returning the value that you have specified.

ferror Flavor

This flavor is a mixture of **error**, **sys:no-action-mixin**, and **sys:proceed-with-value-mixin**. It is the default flavor used by the functions **error** and **ferror** and is often convenient for programs to instantiate. The **ferror** flavor is a good generic error condition where the error message is more important than the proceed types. Signaling the condition **nil** or an unknown condition name makes a condition instance of this flavor.

sys:warning Flavor

This flavor is a mixture of **sys:no-action-mixin** and **condition**. Its message prefix is `>>Warning`. Since the **sys:warning** flavor is not built on **error**, it never invokes the debugger.

break Flavor

This flavor makes a condition a break condition. The error message prefix for **break** is `>>Keyboard`. When you press META-BREAK or META-CTRL-BREAK on the keyboard, this condition is signaled, and the debugger is invoked. The **break** flavor provides the **:no-action** proceed type to simply proceed.

sys:abort Condition

This condition is signaled when the ABORT key is pressed. When signaled while in the debugger, it aborts the current operation and returns control to the innermost command loop in the debugger. Usually, **sys:abort** is used in the **error-restart** special form to return to the innermost command loop in a program when no handler handles it. (For information about **error-restart**, see paragraph 20.4.4, Nonlocal Proceed Types.)

The following are error conditions that the evaluator can signal or that can be signaled by calls to compiled functions. This information is for those who are writing condition handlers. For convenience, the definitions of conditions in this paragraph are followed by descriptions of the condition(s) on which they are based. The novice should skip this information.

sys:invalid-form (error) Condition

This condition is signaled when **eval**'s argument is not a recognizable kind of form: the wrong data type, perhaps. The condition instance supports the operation **:form**, which returns the form with the problem.

sys:invalid-function (error) Condition

This condition is signaled when an object to be applied to arguments is not a valid Lisp function. The condition instance supports the operation **:function**, which returns the supposed function to be called. The **:new-function** proceed type is provided; it expects one argument, a function to call instead of the invalid function.

sys:invalid-lambda-list (**sys:invalid-function error**) Condition

This condition name is present in addition to **sys:invalid-function** when the function to be called looks like an interpreted function and the only problem is the syntax of its lambda list.

sys:too-few-arguments (error) Condition

This condition is signaled when a function is applied to too few arguments. The condition instance supports the operations **:function** and **:arguments**, which return the function and the list of the arguments provided. The proceed types **:additional-arguments** and **:new-argument-list** are provided. Both take one argument. In the first case, the argument is a list of arguments to pass in addition to the ones supplied. In the second case, it is a list of arguments to replace the ones actually supplied.

sys:too-many-arguments (error) Condition

This condition is similar to **sys:too-few-arguments**. Instead of the **:additional-arguments** proceed type, **:fewer-arguments** is provided. Its argument is the number of the originally supplied arguments to use in calling the function again.

sys:undefined-keyword-argument (error) Condition

This condition is signaled when a function that takes keyword arguments is given a keyword that it does not accept (if &allow-other-keys was not used in the function's definition and **:allow-other-keys** was not specified by the caller). The **:keyword** operation on the condition instance returns the extraneous keyword, and the **:value** operation returns the value supplied with it. The proceed type **:new-keyword** is provided. It expects one argument, which is a keyword to use instead of the one supplied.

sys:cell-contents-error (error) Condition

This condition name categorizes all the errors signaled because of unexpected objects found in memory. It includes *unbound* variables, *undefined* functions, and bad data.

This condition supports the following operations:

:address — A locative pointer to the referenced cell.

:current-address — A locative pointer to the cell currently containing the contents that were found in the referenced cell when the error happened. This can be different from the original address in the case of dynamic variable bindings, which move between special PDLs and symbol value cells.

:cell-type — A keyword indicating what type of cell was referenced: **:function**, **:value**, **:closure**, **:instance**, or **nil** or **:unknown** (for a cell that is not one of the preceding kinds).

:containing-structure — The object (list, array, symbol) inside which the referenced memory cell is found.

:data-type
:pointer — The data type and pointer fields of the contents of the memory cell at the time of the error. Both are fixnums.

The proceed type **:no-action** takes no argument. If the cell's contents are now valid, the program proceeds, using them. Otherwise, the error happens again.

The proceed type **:package-dwim** looks for symbols with the same name in other packages but only if the containing structure is a symbol. (The term *dwim* stands for *do what I mean.*)

Two other proceed types take one argument: **:new-value** and **:store-new-value**. The argument is used as the contents of the memory cell. The argument **:store-new-value** also permanently stores the argument into the cell.

sys:unbound-variable (**sys:cell-contents-error**) Condition

This condition name categorizes all errors of variables that are unbound.

sys:unbound-symbol (**sys:unbound-variable**) Condition
sys:unbound-closure-variable (**sys:unbound-variable**) Condition
sys:unbound-instance-variable (**sys:unbound-variable**) Condition

These condition names appear in addition to **sys:unbound-variable** to subcategorize the kind of variable reference in which the error occurred.

sys:undefined-function (**sys:cell-contents-error**) Condition

This condition name categorizes errors of function specs that are undefined.

sys:wrong-type-argument (**error**) Condition

This condition is signaled when a function checks the type of its argument and rejects it; for example, if you try to evaluate `(car 2)`.

The condition instance supports these extra operations:

:arg-name — The name of the argument that was erroneous. This may be **nil** if there is no name or if the system no longer remembers which argument it was.

:old-value — The value that was supplied for the argument.

:function — The function that received and rejected the argument.

:description — A symbol indicating what sort of object was expected for this argument. This symbol is not necessarily a type name. For errors signaled by microcode, it is usually present in **eh:*data-type-names***.

The proceed type **:argument-value** is provided; it expects one argument, which is a value to use instead of the erroneous value.

sys:throw-tag-not-found (eh:trap-throw-error) Condition

This condition is signaled when **throw** is used and no **catch** exists for the specified tag. The condition instance supports these extra operations:

:tag — The tag thrown to.

:values — The values thrown (the values of the second argument to **throw**) as a list.

The proceed type **:new-tag** expects one argument—a tag to throw to instead of the one being thrown to.

Basic Condition Operations

20.5.2 All condition flavors include the following methods, which are normally used by condition handlers.

:condition-names Method of **condition**

This method returns a list of all the condition names for this condition instance. This list includes the name of the flavor that was actually signaled, all of its component flavors, its components' components, and so on, plus other nonflavor condition names attached to this signal.

:report *stream* Method of **condition**

This method prints on *stream* the condition's error message: a description of the circumstances for which the condition instance was signaled. The output should not start or end with a carriage return.

If you are defining a new flavor of condition and wish to change the way the error message is printed, this is the method to redefine. All others use this one.

Every condition instance can print an *error message* that describes the circumstances leading to the signaling of the condition. The easiest way to print one is to print the condition instance without slashification by using such functions as **princ**, or **format** with ~A. These functions actually use the **:report** operation, which implements the printing of an error message. When a condition instance is printed with slashification, it uses the #⊏ syntax so that it can be read back in.

:report-string Method of **condition**

This method returns a string containing the text that the **:report** method prints.

Condition handlers use the next two methods to prevent handling of certain types of errors.

:dangerous-condition-p Method of **condition**

This method returns true if the condition instance is one of those that indicate events considered extremely dangerous, such as running out of memory. Handlers that normally handle all conditions should possibly make an exception for these. One such condition is **virtual-memory-overflow**.

:debugging-condition-p Method of **condition**

This method returns true if the condition instance contains a condition that is signaled for debugging. For example, **break** is signaled when you type META-BREAK or META-CTRL-BREAK. These conditions are not errors although they normally enter the debugger. Any condition handler that is defined to handle *all* conditions should probably make a specific exception for these. Conditions used for debugging include **break**, **mar-break**, **step-break**, **breakpoint**, **trace-breakpoint**, **exit-trap**, and **call-trap**.

See also the methods **:proceed**, **:proceed-asking-user**, **:proceed-types**, **:proceed-type-p**, **:user-proceed-types**, **:document-proceed-types**, which all deal with proceeding (in paragraph 20.4.2, Proceeding and the Debugger).

Condition Methods Used by the Debugger

20.5.3 Some methods are intended for the debugger to use. They are documented because some flavors of **condition** redefine them, causing the debugger to behave differently. They are not often needed by users.

:print-error-message *stack-group brief-flag stream* Method of **condition**

This method is used by the debugger to print a complete error message. It uses the **:report** and **:print-error-message-prefix** methods.

Certain condition flavors define an **:after :print-error-message** method that, when *brief-flag* is **nil**, prints additional helpful information that is not part of the error message per se. Often this operation requires access to the erring stack group in addition to the data in the condition instance. The method can assume that if *brief-flag* is **nil**, then *stack-group* is not the one executing.

For example, the condition signaled when you call an undefined function has an **:after :print-error-message** method that checks for the case of calling a function such as **bind** that is meaningful only in compiled code; if this is what happens, it searches the stack to look for the name of the function in which the call appears. This information is not considered crucial to the error itself and is therefore not recorded in the condition instance.

:print-error-message-prefix Method of **condition**

This method prints an error message prefix, such as `>>Condition`, `>>Error`, `>>Trap`, `>>Warning`, or `>>Keyboard`.

:maybe-clear-input *stream* Method of **condition**

This method is used on entry to the debugger to discard input. Certain condition flavors used by stepping redefine this method to do nothing, so the input is not discarded. These condition flavors include **step-break**, **breakpoint**, **trace-breakpoint**, **throw-exit-trap**, **exit-trap**, and **call-trap**.

:bug-report-recipient-system Method of **condition**

The CTRL-M debugger command uses this operation in determining the address to which bug reports are mailed. By default, it returns **"LISPM"**. The value returned by this method is passed as the first argument to the function **bug**.

:bug-report-description *stream* &optional *numeric-arg* Method of **condition**

This method is used by the debugger's CTRL-M command to print on *stream* the information that should go in the bug report mail buffer. The optional argument *numeric-arg* is what the user gives to the CTRL-M command. It specifies the number of frames from the backtrace to print verbosely on *stream*.

:find-current-frame *stack-group* Method of **condition**

This method returns the stack indices of the stack frames on which the debugger should operate. (See Section 26, Stack Groups, for information about frames.) It returns four values.

The first value is the frame at which the error occurred. This is not the innermost stack frame; it is outside the calls to such functions as **ferror** and **signal-condition**, which are used to signal the error.

The second value is the initial value for the current frame when the debugger is first entered.

The third value is the innermost frame that the debugger should be willing to let the user see. By default this is the innermost stack frame.

The fourth value, if non-**nil**, tells the debugger to consider the innermost frame to be *interesting*. Normally, frames that are part of the interpreter (calls to ***eval**, **sys:apply-lambda**, **prog**, **cond**, and so on) are considered uninteresting.

:debugger-command-loop *stack-group* &optional *error-object* Method of **condition**

This method enters the debugger's command loop. The initial error message and backtrace have already been printed. This method is used by an error handler stack group. The *stack-group* argument specifies the stack group in which the condition was signaled. The *error-object* argument is the condition instance that was signaled; it defaults to the instance to which this message is sent.

The **:debugger-command-loop** method uses the **:or** method combination (see Section 19, Flavors). Some condition flavors add methods that perform another sort of processing or enter a different command loop. For example, unbound variable errors search for look-alike symbols in other packages at this point. If the added method returns **nil**, the original method that enters the debugger's command loop is called.

Creating Condition Instances

20.5.4 You can create a condition instance with **make-instance** if you know which instance variables to initialize. For example:

```
(make-instance 'ferror :condition-names '(foo)
                       :format-string "~S loses."
                       :format-args losing-object)
```

This code creates an instance of **ferror** just like the one that is signaled by the following form:

```
(ferror 'foo "~S loses." losing-object)
```

Note that the condition name for the condition instance is set to what is specified for the **:condition-names** keyword in addition to the flavor name and its components' names.

Direct use of **make-instance** is cumbersome, however, and it is usually easier to define a signal name with **defsignal** or **defsignal-explicit** and then create the instance with **make-condition**.

The signal name is an abbreviation for all the items that are always the same for a certain type of condition: the flavor to use, the condition names, and the arguments that are expected. In addition, it allows you to use a positional syntax for the arguments, which is usually more convenient in simple use than a keyword syntax.

The following is a typical **defsignal**:

```
(defsignal series-not-convergent sys:arithmetic-error (series)
 "Signaled by limit extractor when SERIES does not converge.")
```

This code defines a signal name, `series-not-convergent`, with the flavor name `sys:arithmetic-error`, an interpretation for the arguments `(series)`, and a documentation string. The documentation string is not used in printing the error message; it is documentation for the signal name.

The `series-not-convergent` signal name can then be used to signal an error or merely to create a condition instance:

```
(ferror 'series-not-convergent
        "The series ~S went to infinity."
        myseries)
```

or

```
(make-condition 'series-not-convergent
                "The series ~S went to infinity."
                myseries)
```

The list `(series)` in the **defsignal** is a list of implicit instance variable names. They are matched against arguments to **make-condition** following the format string. Each implicit instance variable name becomes an operation defined on the condition instance to return the corresponding argument passed to **make-condition**. (You can imagine that **:gettable-instance-variables** is in effect for all the implicit instance variables.) In this example, sending a **:series** message to the condition instance returns the value specified via *myseries* when the condition was signaled. The implicit instance variables are actually implemented using the condition instance's property list.

Thus, **defsignal** spares you the need to create a new flavor merely to remember a particular piece of information about the condition.

defsignal *signal-name* {*flavor*|(*flavor* {*condition-names*}*)} ({*implicit-instance-variable*}*) &optional *documentation* {*extra-init-keyword-form*}* Macro

This macro defines a function (**:property** *signal-name* **eh:make-condition-function**) to create an instance of *flavor* with *condition-names*. It also creates implicit instance variables whose names are taken from the list *implicit-instance-variables* and whose values are taken from the arguments following the format string in **make-condition**. The signal name defined by **defsignal** may be passed as an argument to **ferror**, **cerror**, **error**, and **signal**.

If just a flavor name is specified instead of a list containing a flavor name and condition names, this flavor name is equivalent to using *signal-name* as the sole condition name.

The *extra-init-keyword-forms* are forms to be evaluated to produce additional keyword arguments to pass to the **make-instance** of **flavor**. These can be used to initialize other instance variables that particular flavors may have. These expressions can refer to the *implicit-instance-variables*.

For the previous example, the `defsignal` for `series-not-convergent` creates the following function:

```
(defun (:property series-not-convergent eh:make-condition-function)
 (signal-name format-string &optional series &rest format-args)
 (declare (ignore signal-name))
 (make-instance 'arithmetic-error
               :property-list (list :series series)
               :format-string format-string
               :format-args (list* series format-args)
               :condition-names '(series-not-convergent)))
```

defsignal-explicit *signal-name* *{flavor|(flavor {condition-names}*)}* *signal-arglist documentation {init-keyword-forms}** Macro

Like **defsignal**, this macro defines a signal name. This signal name is used in the same way, but it creates the condition instance differently.

First, there is no list of implicit instance variables. Instead, *signal-arglist* is a lambda list that is matched against all the arguments to **make-condition** except for the signal name itself. The variables bound by the lambda list can be used in the *init-keyword-forms*, which are evaluated to enable arguments to pass to **make-instance**. For example:

```
(defsignal-explicit mysignal-3
            (my-error-flavor mysignal-3 my-signals-category)
            (format-string losing-object &rest format-args)
   "The third kind of thing I like to signal."
   :format-string format-string
   :format-args (cons losing-object format-args)
   :losing-object-name (send losing-object :name))
```

Since implicit instance variables are merely properties on the property list of the instance, you can create them by using the **:property-list** initialization keyword. The contents of the property list determine which implicit instance variables there are and their values.

For the previous example, the `defsignal-explicit` for `mysignal-3` creates the following function:

```
(defun (:property mysignal-3 make-condition-function)
      (signal-name format-string losing-object &rest format-args)
   "The third kind of thing I like to signal"
   (declare (function-parent mysignal-3))
   (declare (ignore signal-name))
   (make-instance 'my-error-flavor
                 :format-string format-string
                 :format-args (cons losing-object format-args)
                 :losing-object-name (send losing-object :name)
                 :condition-names '(mysignal-3 my-signals-category)))
```

make-condition *signal-name* &rest *condition-arguments* Function

This function is the fundamental way that condition instances are created. The *signal-name* indicates how to interpret the *condition-arguments* and come up with a flavor and values for its instance variables. The handling of the *arguments* is entirely determined by the *signal-name*.

If *signal-name* is a condition instance, **make-condition** returns it. It is not useful to call **make-condition** explicitly in this way, but this procedure allows condition instances to be passed to the convenience functions **error** and **signal**, which call **make-condition**.

If *signal-name* was defined with **defsignal** or **defsignal-explicit**, then that definition specifies exactly how to interpret the *condition-arguments* and to create the instance. In general, if *signal-name* has an **eh:make-condition-function** property (which is how **defsignal** works), this property is a function to which *signal-name* and *condition-arguments* are passed, and it does the work.

Alternatively, *signal-name* can be the name of a flavor. Then the *condition-arguments* are passed to **make-instance**, which interprets them as initialization keywords and values.

If *signal-name* has no **eh:make-condition-function** property and is not a flavor name, then this trivial **defsignal** is assumed as a default:

`(defsignal` *signal-name* `ferror ())`

In this case, the value of **make-condition** is an instance of **ferror**, with *signal-name* as a condition name, and the *condition-arguments* are interpreted as the format string and format arguments.

The *signal-name* **nil** actually has a definition of this form and is frequently used as the signal name when you do not want to use any particular condition name. When **error** and **ferror** have no *signal-names*, they actually signal **nil**.

Signaling a Condition Instance

20.5.5 Once you have a condition instance, you are ready to invoke the condition handling mechanism by signaling it. A condition instance can be signaled any number of times, in any stack group.

signal-condition *condition-instance* &optional *proceed-types* Function
invoke-debugger ucode-error-status inhibit-resume-handlers

This function invokes the condition handling mechanism on *condition-instance*, possibly enters the debugger, and/or possibly proceeds or resumes.

The *proceed-types* argument is a list of proceed types (among those conventionally defined for the type of condition you have signaled) that you are prepared to implement if a condition handler returns one (see paragraph 20.4, Proceeding). These *proceed-types* are also referred to as *local proceed types* and are in addition to any proceed types implemented nonlocally by the **condition-resume** macro (as described in paragraph 20.4.4, Nonlocal Proceed Types).

The **signal-condition** function returns to its caller only in one of three situations:

- It decides to proceed using one of the *proceed-types*.
- The condition is not an error, and there are no nonlocal proceed types to be used.
- The *inhibit-resume-handlers* argument is non-**nil**.

The *ucode-error-status* argument is used for internal purposes in signaling errors detected by the microcode.

The **signal-condition** function tries various possible handlers for the condition. Each handler that is tried can terminate the act of signaling by throwing out of **signal-condition**, or it can specify a way to proceed from the signal. The handler can also decline to handle the condition by returning **nil**, and then the next possible handler is tried.

First, **eh:*condition-handlers*** is scanned for handlers that are applicable (according to the condition names they specify) to this condition instance. After this list is exhausted, **eh:*condition-default-handlers*** is scanned in the same way.

Finally, if *invoke-debugger* is non-**nil**, the debugger (the handler of last resort) is invoked. With the debugger, you can ask to throw or to proceed. The default value of *invoke-debugger* is non-**nil** if the *condition-instance* is an error.

It is possible for all handlers to decline the condition if the debugger is not among the allowed handlers tried. (The debugger cannot decline to handle the condition.) In this circumstance, **signal-condition** proceeds using the first proceed type on the list of available ones, provided that it is a nonlocal proceed type. If it is a local proceed type or if there are no proceed types, **signal-condition** simply returns **nil**. (It would be slightly simpler to proceed using the first proceed type whether it is local or not. But in the case of a local proceed type, this simply means returning the proceed type instead of **nil**. It is considered slightly more useful to return **nil**, allowing the signaler to distinguish the case of a condition not handled. The signaler knows which proceed types it specified and can easily consider **nil** as equivalent to the first of them if it wants to.)

Otherwise, by this stage, a proceed type has been chosen from the available list. If the proceed type was among those specified by the caller of **signal-condition**, then proceeding consists simply of returning to that caller. The chosen proceed type is the first value, and arguments (returned by the handler along with the proceed type) may follow it. If the proceed type was implemented nonlocally with **condition-resume** (discussed in paragraph 20.4.4, Nonlocal Proceed Types), then the associated proceed handler function in **eh:*condition-resume-handlers*** is called.

If *inhibit-resume-handlers* is non-**nil**, resume handlers are not invoked. If a handler returns a nonlocal proceed type, **signal-condition** simply returns to its caller as if the proceed type were local. If the condition is not handled, **signal-condition** returns **nil**.

The **condition-bind-default** macro allows you to define a handler that handles an error only if it is not handled by any of the callers' handlers. (This form is described in paragraph 20.3, Handling Conditions.) A more flexible technique for performing this type of operation is to make the condition handler signal the same condition instance recursively by calling **signal-condition**, such as in the following:

```
(multiple-value-list
  (signal-condition condition-instance
            eh:*condition-proceed-types* nil nil t))
```

A handler that uses this technique passes along the same list of proceed types specified by the original signaler, prevents the debugger from being called, and prevents resume handlers from being run. If the first value that **signal-condition** returns is non-**nil**, then one of the outer handlers has handled the condition. Your handler's simplest option is to return those same values so that the other handler has its way (but you can also examine them and return modified values). Otherwise, you go on to handle the condition in your default manner.

eh:*trace-conditions* Variable

This variable can be set to a list of condition names to be *traced*.

Whenever a condition possessing a traced condition name is signaled, an error is signaled to report the fact before any handler is called. (Tracing of conditions is turned off when this error is signaled.) Proceeding with the proceed type **:no-action** causes the signaling of the original condition to continue.

If **eh:*trace-conditions*** is the symbol **t**, all conditions are traced.

21

COMPILER OPERATIONS

Introduction

21.1 The Lisp compiler converts Lisp functions into code in the Explorer system's instruction set so that they run more quickly and take up less storage. Compiled functions are maintained as Lisp objects of type **compiled-function,** which contain machine code and other information. If you want to understand the output of the compiler, see Section 22, The Disassembler.

Invoking the Compiler

21.2 If the **compile, compile-lambda, compile-file,** and **compile-form** compiler interface functions are called so that they return two values, such as with **multiple-value-bind** or **multiple-value-setq,** the second value returned is a status value indicating the worst thing that happened during compilation.

compiler:ok	Constant
compiler:warnings	Constant
compiler:errors	Constant
compiler:fatal	Constant
compiler:aborted	Constant

The compiler status value is a number equal to one of these constants. They have the following meanings:

- **compiler:ok** — No problems were found.
- **compiler:warnings** — Warning messages were issued.
- **compiler:errors** — Error messages were issued.
- **compiler:fatal** — Fatal errors were encountered that prevented generation of an object for one or more of the functions.
- **compiler:aborted** — The compilation was interrupted; no object was generated.

These values are ordered so that **compiler:ok** is the smallest, and **compiler:aborted** is the largest.

For example, you can have a file compiled and then loaded with the stipulation that nothing worse than warnings can occur during compilation. In this case, the code appears something like this:

```
(multiple-value-bind (outfile status)
    (compile-file "filename")
  (when (< status compiler:errors)
    (load outfile)
           ))
```

The following functions are used to invoke the compiler.

compile *name* &optional *definition* *[c]* Function

This function can be used to compile Lisp functions.

If *definition* is supplied, it should be a lambda expression. Otherwise, *name* (usually a symbol but possibly a more general function specification as defined in Section 16, Functions) should be defined as an interpreted function, and its definition is used as the lambda expression to be compiled. The compiler converts the lambda expression into a compiled function, saves the lambda expression as the **:previous-definition** property of *name* (if *name* is a symbol), and changes *name*'s definition to be the new compiled function. For information about **fdefine**, see Section 16, Functions.

If *function-spec*'s definition is already a compiled function and this compiled function's debugging information remembers the interpreted definition it was compiled from, the same definition is compiled again. The original definition is recorded in a compiled function's debugging information whenever the function is compiled in memory (such as by means of **compile**) but not if the function is loaded from an object file, except for **defsubsts** or functions proclaimed **inline**.

If *name* is **nil**, then you must supply *definition*; the resulting FEF is returned as the value of the call to **compile**. The **compile-lambda** function is preferable for this purpose since it allows you to specify a name for the compiled function.

uncompile *function-spec* *[c]* Function
uncompile *function-spec* &optional *dont-unencapsulate* Function

If *function-spec* is defined as a compiled function that records the original definition that was compiled, then *function-spec* is redefined with that original definition. This cancels the effect of calling **compile** on *function-spec*.

If *function-spec* is not so defined, an error message is returned. If *dont-unencapsulate* is true, then **sys:unencapsulate-function-spec** is not applied to *function-spec* prior to uncompiling the function.

compile-lambda *lambda-exp function-spec* Function

This function returns a compiled function object produced by compiling *lambda-exp*. The function name recorded in the compiled function object is *function-spec*; however, *function-spec* is not defined by **compile-lambda**.

compile-encapsulations *function-spec* Function

This function compiles all encapsulation that **function-spec** currently has. Encapsulations include tracing, breakons, and advice. (For information about encapsulations, see Section 16, Functions.) Compiling traces or breakons makes it possible (or at least more possible) to trace or break on certain functions used in the evaluator. Compiling advice makes it less costly to advise functions that are used frequently.

Any encapsulation that is changed ceases to be compiled; thus, if you add or remove advice, you must execute **compile-encapsulations** again if you wish the advice to be compiled again.

compile-encapsulations-flag Variable

If this variable is non-**nil** (it is initially **nil**), all created encapsulations are compiled automatically.

compile-file *filename* &key **:output-file** *[c]* Function
compile-file *filename* &key **:output-file :load :verbose** Function
:set-default-pathname :package :declare :suppress-debug-info

This function translates a file containing Lisp source code into an object file that describes the compiled functions and associated data. The object file format is capable of representing an arbitrary collection of Lisp objects, including shared structures and cycles of pointers. The function returns two values: the pathname object of the output file and the status.

Macro definitions, **subst** definitions, and **special** declarations created during the compilation are canceled when the compilation is finished.

filename — The *filename* argument must specify a source file (written in Lisp), either by a pathname object, namestring, or stream object. The variable ***default-pathnames-defaults*** provides the defaults for *filename*.

:output-file — This keyword allows you to specify a file pathname as an output file. The default value is the same as *filename*, except that the file type is xld.

The following options are Explorer extensions.

:load — If **:load** is true, the compiler loads the output file after compilation (assuming no fatal errors occurred during compilation).

:verbose — If **:verbose** is true, the compiler prints the name of each function as it begins compiling that function. Printing goes to ***standard-output***. The **:verbose** keyword defaults to the value of the **compiler:compiler-verbose** variable, which is normally **nil**.

If you supply both the **:load** and **:verbose** arguments, then **compile-file** also passes the **:verbose** argument to the loader.

:set-default-pathname — If **:set-default-pathname** is true, the compiler updates the pathname default from the supplied *filename*.

:package — This keyword allows you to specify the package in which the file is to be compiled rather than accepting the value supplied in the file attribute line (-*-) of that file.

:declare — The value of this keyword is either a declaration specifier or a list of declaration specifiers. The compiler processes these declaration specifiers as if they appeared in a **proclaim** form at the beginning of the source file. The **:declare** keyword is most useful for setting optimization levels. For example:

```
(compile-file "source" :declare '(optimize speed))
```

:suppress-debug-info — When this option is **t**, the compiler does not record debug information in the object file. This information includes the documentation string, local variable map, argument names, and local function names. However, if the name of the function is an external symbol, then the argument list and documentation string are recorded anyway. If the value of **:suppress-debug-info** is **:documentation**, then only the documentation string is suppressed.

The motivation for this option is to be able to hide internal information about proprietary programs, making it more difficult for someone to figure out the program by studying the object code. This option also reduces band size and increases compilation and loading speed.

compiler:*output-version-behavior* Variable

This variable controls the version number picked by **compile-file** for output files. The acceptable values for this variable and their meanings are as follows:

:same — The output file has the same version as the source file.

:newest — The output file has a version number one higher than the previous highest version output file.

:higher — As with **:same**, the output file has the same version as the source file unless there is already a file with the same or higher version number (a *collision*), in which case, as with **:newest**, the next higher version number is used. Note that this option is somewhat slower.

:ask-higher — This option is like **:same** but asks the user what to do if there is a collision. If the user does not respond, the next higher version number is used, as in **:newest**.

:ask-same — This option is like **:ask-higher** but if the user does not respond, the output file has the same version as the source file, as with **:same**.

For **:ask-higher** and **:ask-same** you are asked the following:

```
Output file <some file name> already exists.
What would you like to do? (S, N, P, D)
```

where S means **:same**, N means **:higher**, P means ask for a new pathname, and D means to take the time-out default. The default is either **:same** or **:higher** depending on whether you used **:ask-same** or **:ask-higher**.

The **make-system** function also checks this variable to see whether to compare version numbers or creation dates when deciding whether the source is newer than the object. However, if the source file is on a remote file server that does not support version numbers, then object files are written with the version **:newest** regardless of this variable. Also, if **compile-file** is given an **:output-file** pathname that specifies a version, then that version takes preference.

compiler:compile-form *form* Function

This function is similar to the **eval** function in that the argument is evaluated and the resulting value is returned. However, if the *form* has the effect of defining a function, then the function is compiled. Thus, this function can be used to compile function-defining forms such as **defun**, **defstruct**, **defmethod**, **defmacro**, and so on.

Input to the Compiler

21.3 The purpose of **compile-file** is to produce a translated version that does the same thing as the original except that the functions are compiled. The **compile-file** function reads through the input file, processing the forms in it one by one. For each form, suitable binary output is sent to the object file so that when the object file is loaded, the effect of the source form is reproduced. An object file differs from a source file in two ways. First, object files are in a compressed binary form, which reads much faster but cannot be edited. Second, function definitions in object files have been translated from Lisp forms to compiled function objects.

The compiler can handle top-level forms in a file one of three ways. If the form simply defines a function or stores a value into a special variable, then special commands are placed in the object file to perform those operations. For other arbitrary forms, the compiler usually places a command in the object file that calls **eval** to evaluate a list that might be either the original source form or a macro-expanded and optimized equivalent. For more complicated forms (such as those containing definitions of functions as lexical closures), the compiler may compile the form as a function with no arguments and cause the compiled function to be called when the file is loaded.

compiler:compiler-verbose Variable

If this variable is non-**nil**, the compiler prints the name of each function that it is about to compile. The default value is normally **nil**.

compiler:peep-enable Variable

Peephole optimizing is a procedure in which the compiler looks in code for patterns that it knows how to optimize into a more efficient form. This variable is set to true by default to enable this optimizing procedure. Set this variable to **nil** only if you suspect a bug in the optimizer.

compiler:warn-on-errors Variable

If this variable is non-**nil**, errors in reading code to be compiled and errors in macro expansion within the compiler produce only warnings; they do not enter the debugger. The variable is normally **t**.

The default setting is useful when you do not anticipate errors during compilation, because it allows the compilation to proceed past such errors. If you have walked away from the machine, you do not come back to find that your compilation stopped in the first file and did not finish.

If you find an inexplicable error in reading or macro expansion and wish to use the debugger to localize it, set **compiler:warn-on-errors** to **nil** and recompile.

compiler:*warn-of-superseded-functions-p* Variable

If, when you are compiling a file in Common Lisp mode, this variable is true, then the compiler warns about the use of Zetalisp functions that have been superseded by new Common Lisp functions. The default value is **nil**.

Sometimes you need to put items into the file that are not merely meant to be translated into object file form. For example, top-level macro definitions must actually be defined within the compiler for the compiler to be able to expand them at compile time. So when a macro form is seen, usually it should be evaluated at compile time as well as put into the object file.

You might also want to put compiler declarations in a file. These are forms that should be evaluated at compile time to pass certain information to the compiler.

Therefore, a facility exists that allows you to tell the compiler exactly what to do with a form. For instance, you might want a form to be treated in one of the following ways:

- Compiled and put into the object file
- Compiled but not put into the object file
- Evaluated within the compiler
- Not evaluated within the compiler
- Evaluated if the file is read directly into Lisp
- Not evaluated if the file is read directly into Lisp

The **eval-when** special form is used to effect this type of control. An **eval-when** form looks like the following:

```
(eval-when times-list
 form1
  form2
...)
```

The *times-list* may contain one or more of the **load**, **compile**, or **eval** symbols. If **load** is present, the forms are written into the object file to be evaluated when the object file is loaded (except for **defun** forms that put the compiled definition into the object file instead). If **compile** is present, the forms are evaluated in the compiler. If **eval** is present, the forms are evaluated when read into Lisp, because **eval-when** is defined as a special form in Lisp. (The compiler ignores **eval** in the times-list.) For example, the following form would define `foo` as a macro in the compiler when the file is read in and interpreted, but not when the object file is fasloaded:

```
(eval-when (compile eval)
  (defmacro foo (x)
    (cadr x)))
```

When you are compiling from an editor buffer into memory, either **compile** or **load** enables evaluation because compilation into memory is like compiling and loading at the same time.

When the interpreter sees **eval-when**, and one of the *times* is the **eval** symbol, then the *body* forms are evaluated; otherwise, **eval-when** does nothing. But when seen by the compiler, **eval-when** performs the special operations described previously.

For the rest of this section, lists such as those given to **eval-when**, that is, (**load eval**), (**load compile**), and so on, are used to describe when forms are evaluated.

If a form is not enclosed in an **eval-when**, then the time at which it is evaluated depends on the form. The following table summarizes the times when evaluation takes place for the specified form seen at top level by the compiler.

Table 21-1 When Evaluation Occurs With the Compiler

Form	Occurrence
(**eval-when** *times-list form* ...)	*times-list* specifies when the *form*... should be performed.
(**proclaim** ...)	**proclaim** is performed at (**load compile eval**) time, except for **optimize** declarations, which are performed at (**compile eval**) time.
(**declare** (**special** ...)) or (**declare** (**unspecial** ...))	**special** or **unspecial** is performed at (**load compile**) time. (This usage is obsolete.)
(**declare** *anything-else*)	*anything-else* is performed only at (**compile**) time. (This usage is obsolete.)
(**special** ...) or (**unspecial** ...)	(**load compile eval**). (This usage is obsolete.)
(**macro** ...) or (**defmacro** ...) or (**defsubst** ...) or (**defflavor** ...) or (**defconstant** ...) or (**defstruct** ...) or (**deftype** ...)	(**load compile eval**). However, during file-to-file compilation, the definition is kept in effect only during that compilation. The specified form is reexecuted at load time.
(**comment** ...)	Ignored at all times.
(**compiler-let** ((*var val*) ...) &body *body*	Processes the *body* in its normal fashion, but at (**compile eval**) time, the indicated variable bindings are in effect. These variables typically affect the operation of the compiler or of macros.
(**local-declare** (*decl decl* ...) *body* ...)	Processes the *body* in its normal fashion with the indicated declarations added to the front of the list that is the value of **local-declarations**.
(**defun** ...) or (**defmethod** ...) or (**defselect** ...)	(**load eval**); at load time, this form itself is not processed, but the result of compiling it is processed.

The following forms default to (**load compile eval**):

make-package
import
shadowing-import
unuse-package
in-package
require
unexport
defpackage
export
shadow
use-package

Any other forms default to (**load eval**).

Sometimes a macro tries to return more than one form for the compiler top level to see (and to be evaluated). These forms should be nested in a **progn** form. If a **progn** form is seen at the compiler top level, all of the forms are processed as if they also had been at compiler top level.

dont-optimize *form* Special Form

To prevent an expression from being optimized by the compiler, surround it with a call to this special form.

In execution, this special form is equivalent to simply *form*. However, any source-level optimizations that the compiler normally performs on the top level of *form* are not done. For example:

```
(dont-optimize (apply 'foo (list 'a 'b)))
```

This form actually makes a list and calls `apply`, rather than doing the following:

```
(foo 'a 'b)
```

Similarly, the following code actually calls `sys:flavor-method-table` as a function, rather than substituting the definition of that **defsubst**:

```
(dont-optimize (sys:flavor-method-table flav))
```

The **dont-optimize** special form can even be used around a **subst** or a function declared **inline** inside of **setf** or **locf** to prevent open coding of the **defsubst**. In this case, a function is created at load time to do the setting or to return the location:

```
(setf (dont-optimize (zwei:buffer-package buffer))
      (find-package "foo"))
```

Subforms of *form*, such as arguments, are still optimized or open coded, unless additional **dont-optimize**'s appear around them.

Precompilation Considerations

21.4 Explorer utilities such as Zmacs assume that source files are formatted so that an opening parenthesis at the left margin (that is, in the first column) indicates the beginning of a function definition or other top-level list, with a few standard exceptions. The compiler assumes that you follow this indentation convention, enabling it to tell when a closing parenthesis is missing from one function as soon as the beginning of the next function is reached.

If the compiler finds an opening parenthesis in the first column in the middle of a list, it invents enough closing parentheses to close off the list that is in progress. A compiler warning is produced instead of an error. After this list has been processed, the opening parenthesis is read again. The compilation of the list that was forcefully closed off is probably useless, but the compilation of the rest of the file is usually correct. You can read the file into the editor, fix the error, and recompile only the function that was unbalanced.

Certain special forms, including **eval-when**, **progn**, **declare-flavor-instance-variables**, and **comment**, are customarily used around lists that start in the first column. These symbols have a non-**nil** **sys:may-surround-defun** property that makes the compiler permit this convention. You can add such properties to other symbols if you choose to do so.

compiler:qc-file-check-indentation Variable

The compiler checks for opening parenthesis in the first column if this variable is non-**nil**. (It is normally **t**.)

Some advertised variables have compile-time values that affect the operation of the compiler. The only meaningful way that users can set these variables is by a global **setq**, by a **compiler-let**, or by including in their files forms such as the following:

```
(eval-when (compile) (setf open-code-map-switch t))
```

However, these variables seem not to be needed very often.

obsolete-function-warning-switch Variable

If this variable is non-**nil**, the compiler tries to warn the user whenever an obsolete function is used. The default value is **t**.

open-code-map-switch Variable

If this variable is non-**nil**, the compiler attempts to produce inline code for the mapping functions (**mapc**, **mapcar**, and so on, but not **mapatoms**) if the function being mapped is an anonymous lambda expression. The generated code is also more efficient. The default value is **t**.

inhibit-style-warnings-switch Variable

If this variable is non-**nil**, all compiler style-checking is turned off. Style-checking is used to issue obsolete function warnings, won't-run-in-MacLisp warnings, and other types of warnings. The default value is **nil**. See also the **inhibit-style-warnings** macro in paragraph 21.7, Controlling Compiler Warnings. The macro acts on only one level of an expression.

Compiling From Zmacs

21.5 The Zmacs editor allows you to compile regions, buffers, and even files. The following paragraphs briefly describe how to perform these various compilations; see the *Explorer Zmacs Editor Reference* manual for full details.

Compiling a Region

21.5.1 Zmacs allows you to compile a region in two different ways. The first and easiest way is to bring up the Zmacs menu by clicking right once on the mouse. One of the commands available on this menu is Compile Region. Select this command with the mouse, and Explorer then compiles the region you have marked.

One of the simplest ways to compile a region is to press CTRL-SHIFT-C after selecting the region to be compiled.

The compiled code is now the current definition in memory; thus, if you execute a program that calls this piece of code, the program uses values set during the compilation. These values remain in effect until they are rebound by you or by other code.

Compiling a Buffer

21.5.2 The Compile Buffer command performs an incremental compile and load of each top-level form. As a result, code at the end of the buffer can use definitions occurring at the beginning of the buffer.

To compile a Zmacs buffer, enter the Compile Buffer extended command from the Zmacs minibuffer. As with compiled regions, the compiled buffer is now the current definition in memory. If you execute a program that calls this piece of code, the program uses values bound during the compilation.

A useful keystroke for compiling a buffer is META-Z. When you press META-Z, the buffer is automatically compiled, and then control is returned to whichever utility you were in before entering the editor.

Compiling a File

21.5.3 To compile a file from Zmacs, enter the Compile File extended command in the Zmacs minibuffer. You are prompted for the name of the file you want to compile.

Note that only those items evaluated at compile time are in memory after compiling a file. To get the same results as compiling a buffer, you typically must load the object file after compiling a file. To load the object file, use the Zmacs Load File command or the **load** function.

Using the Warnings Database

21.6 Although this paragraph describes using the warnings database from the point of view of compiler warnings, you should know that other Explorer utilities, such as Prolog, also use the database.

When the compiler prints warnings, it also records them in a warnings database that is organized by file and by function within each file. Old warnings for previous compilations of the same function are thrown away so that the database contains only warnings that are still applicable. This database can be used to visit, in the editor, the functions that encountered warnings. You can also save the database and restore it later.

You can use three other editor commands to begin visiting the sites of the recorded warnings. They differ only in how they decide which files to look through:

- Edit Warnings or Edit Compiler Warnings — For each file that has any warnings, this command asks whether to edit the warnings for that file.
- Edit File Warnings — This commands reads the name of a file and then edits the warnings for that file.
- Edit System Warnings — This command reads the name of a system and then edits the warnings for all files in that system (for more information, see **defsystem** in Section 23, Maintaining Large Systems).

While the warnings are being edited, the warnings themselves appear in a small window at the top of the editor frame, and the code appears in a large window that occupies the rest of the editor frame. (For more information about editing compiler warnings, see the *Explorer Zmacs Editor Reference* manual.)

As soon as you have finished specifying the file(s) or system to process, the editor proceeds to visit the code for the first warning. From then on, to move to the next warning, use the CTRL-SHIFT-W command. To move to the previous warning, use META-SHIFT-W. You can also switch to the warnings window with CTRL-X O or with the mouse, and move around in that buffer. When you use CTRL-SHIFT-W and there are no more warnings after the cursor, you return to single-window mode.

You can also insert the text of the warnings into any editor buffer by executing the following Zmacs commands:

- META-X Insert File Warnings — This command reads the name of a file and inserts the text for that file's warnings into the buffer after point. The mark is left after the warnings, but the region is not turned on.
- META-X Insert Warnings — This command inserts the text for the warnings (of all files that have warnings) into the buffer after point. The mark is left after the warnings, but the region is not turned on.

You can dump the warnings database into a file and reload it later. Then you can perform a META-X Edit Warnings command in the later session. You dump the warnings with **sys:dump-warnings** and load the file again with **load**. You can also dump warnings with META-X Dump Compiler Warnings and reload them with META-X Load Compiler Warnings. In addition, **make-system** with the **:batch** option writes all the warnings into a file in this way.

sys:dump-warnings *output-file-pathname* &rest *warnings-file-pathname...* Function

This function writes the warnings for the files named in *warnings-file-pathname* (one or more pathname objects or namestrings) into a file named *output-file-pathname*.

Controlling Compiler Warnings

21.7 By controlling the compile-time values of the **run-in-maclisp-switch**, **obsolete-function-warning-switch**, and **inhibit-style-warning-switch** variables (explained previously), you can enable or disable some of the warning messages of the compiler. The following macro is also useful:

inhibit-style-warnings *form* Macro

This macro prevents the compiler from checking style (programming items that are not illegal but that are not advisable to use, such as obsolete functions) on the top level of *form*. Style-checking is still performed on the arguments of *form*. Both obsolete function warnings and won't-run-in-MacLisp warnings are executed by the style-checking mechanism.

For example, the following code does not issue a warning indicating that `plus` is a Zetalisp function:

```
(setq bar (inhibit-style-warnings (plus 1 2)))
```

However, the following code does issue a warning since **inhibit-style-warnings** applies only to the top level of the form inside it (in this case, to the **setq**):

```
(inhibit-style-warnings
    (setq bar (plus 1 2)))
```

Sometimes functions take arguments that they deliberately do not use. Normally, the compiler warns you if your program binds a variable that it never references. To disable this warning for variables that you know you are not going to use, there are three things you can do.

The first (and preferred) method to disable such warnings is to declare the variables to be ignored:

```
(declare (ignore var))
```

In this example, `var` is the name of the particular variable to be ignored.

A second, less preferable method is to name the unused variables **ignore** or **ignored**. The compiler does not object if one of these variables is not used. Furthermore, by special dispensation, it is permissible in a lambda list to have more than one variable with one of these names.

The third, and least preferable method is to use the variable for effect (ignoring its value) at the front of the function. For example:

```
(defun the-function (list fraz-name fraz-size)
 fraz-size   ; This argument is not used.
 ...)
```

The code in this example does have the advantage over the second method in that **arglist** (see Section 16, Functions) returns a more meaningful argument list for the function, rather than returning something with **ignores** in it, but the first method has the same advantage and is more explicit.

The following function is useful for requesting compiler warnings in certain esoteric cases. Normally, the compiler notices whenever any function *x* calls any other function *y*. The compiler makes notes of all these uses and then warns you at the end of the compilation if the function *y* was called but was incorrectly defined in the environment or was neither defined nor declared in the compilation. This convention is usually acceptable, but sometimes the compiler cannot tell that a certain function is being used. Suppose that instead of having *x* contain any forms that call *y*, *x* simply stores *y* away in a data structure. At another location in the program, this data structure is accessed and **funcall** is performed on it. The compiler cannot anticipate that this is going to happen. As a result, the compiler does not notice the function usage, so it does not create a warning message. To make such warnings happen, you can explicitly call the **compiler:function-referenced** function at compile time.

compiler:function-referenced *what by* Function

The *what* argument is a symbol that is being used as a function. The *by* argument can be any function spec.

The **compiler:function-referenced** function must be called at compile time while a compilation is in progress (typically in an `(eval-when (compile) ...)` form). This function tells the compiler that the *what* function is referenced by *by*. If, when the compilation is finished, the *what* function has not been defined, the compiler issues a warning to the effect that *by* referred to the function *what*, which was never defined.

You can also tell the compiler about any function it should consider *defined*:

compiler:compilation-define *function-spec* Function

The *function-spec* argument is marked as defined for the sake of the compiler; calls to this function do not produce warnings.

compiler:make-obsolete *function reason-string* Special Form

This special form declares *function* (not evaluated) to be obsolete. When code calls this function, a compiler warning is issued under the control of **obsolete-function-warning-switch**. The compiler uses this special form to mark as obsolete certain functions that exist but that should not be used in new programs. It can also be useful when maintaining a large system, as a reminder that *function* has become obsolete and usage of it should be phased out. An example of an obsolete-function declaration is the following:

```
(compiler:make-obsolete probef
  "use probe-file with the same arguments instead")
```

The second argument can simply be a symbol that is the name of the new function:

```
(compiler:make-obsolete probef probe-file)
```

compiler:make-variable-obsolete *old-name new-name* Special Form

This function causes the compiler to issue a warning about the use of an obsolete name for a special variable. The *old-name* argument should be a symbol, and *new-name* can be a symbol, form, or string.

Compiler Source-Level Optimizers

21.8 The compiler stores optimizers for source code on property lists, making it easy for the user to add optimizers to these lists.

An optimizer can be used to transform code into an equivalent but more efficient form. For instance, Example A is transformed into Example B, which can be more efficiently compiled, as in the following:

Example A:

```
(eq obj nil)
```

Example B:

```
(null obj)
```

The compiler finds the optimizers to apply to a form by looking for the **compiler:optimizers** property of the symbol that is the car of the form. The value of this property should be a list of optimizers, each of which must be a function of one argument. The compiler tries each optimizer in turn, passing the form to be optimized as the argument. An optimizer that returns the original form unchanged (eq to the argument) has *done nothing*, and the next optimizer is tried. If the optimizer returns anything else, it has *done something*, and the whole process starts over again. Only after all the optimizers have been tried and have done nothing is an ordinary macro definition processed. This convention is used so that the macro definitions, which are seen by the interpreter, can be overridden for the compiler by optimizers. Actually, most of the compiler's optimizers are on the **compiler:post-optimizers** property, but since post-optimizers operate on partially compiled code, it is not practical for users to define their own.

Optimizers should not be used to define new language features because they take effect only in the compiler; the interpreter (that is, the evaluator) does not know about optimizers. Therefore, an optimizer should not change the effect of a form; it should produce another form that performs the same operation, possibly faster, with less memory, or with some other benefit. This is why these forms are called *optimizers*. If you actually want to change the form to perform some other operation, use a macro.

compiler:add-optimizer *function optimizer* &rest *optimized-into...* Special Form

This special form puts *optimizer* on *function*'s optimizers list if it is not there already. The *optimizer* argument is the name of an optimization function, and *function* is the name of the function calls that are to be processed. Neither is evaluated.

The following example also remembers *optimize-into-1*, and so forth, as names of functions that can be called in place of *function* as a result of the optimization:

```
(compiler:add-optimizer function optimizer
             optimize-into-1
             optimize-into-2...)
```

Then **who-calls** of *function* also mentions calls of *optimize-into-1* and so on.

compiler:optimize-pattern *template replacement* &optional *condition* Macro

This macro causes calls that match *template* to be optimized to *replacement*. The *template* looks like a function call form except that each argument is represented by one of the following:

- A type name symbol — The optimization can be performed if the argument is known to always be of that type (this should not be confused with the type that the function expects). Note that **t** can be used to indicate that the argument can be anything.
- A **quote** form — The argument must be that constant value.
- A *#'function* form — The argument can be either *#'function* or *'function*.
- The form (**passes** *p*) — Calls function *p* on the argument form to test whether it is acceptable.

The *replacement* is a list whose first element is the new function name and whose remaining elements indicate the new arguments by one of the following:

- An integer specifies to insert that numbered argument from the original form.
- A **quote** or **function** form is used as the actual argument.

For example, the declaration `(optimize-pattern (foo t list) (bar 2 1))` causes `(foo x (the list y))` to be optimized to `(bar (the list y) x)`.

The optional argument *condition* can be used to specify an additional requirement; this argument is a Lisp expression to be evaluated. When this expression evaluates to **nil**, the optimization is not performed. To avoid the overhead of using the evaluator, you should make this expression a special variable symbol or a function call without any arguments (or a macro that expands to either of these, since the macro expansion is performed only once).

Putting Data in Object Files

21.9 You can make an object file that contains data rather than a compiled program. This option can be useful to speed up the loading of a data structure into the machine, as compared with the reading in of printed representations. Also, certain data structures do not have a convenient printed representation as text but can be saved in object files.

For example, the system stores fonts this way. Each font is in an object file (in the SYS:FONTS; directory) that contains the data structures for that font. When the file is loaded, the symbol that is the name of the font is set to the array that represents the font. Putting data into an object file is often referred to as *fasdumping* the data (because loading a file is sometimes called *fasloading*).

In compiled programs, the constants are saved in the object file in this way. The compiler optimizes programs by converting **equal** constants into **eq** constants when the file is loaded. This step does not happen when you make a data file yourself; the identity of objects is preserved. Note that when an object file is loaded, objects that were **eq** when the file was written are still **eq**; this does not normally happen with source files.

The following types of objects can be represented in object files:

- Symbols
- Numbers of all kinds
- Lists
- Strings
- Arrays of all kinds
- Instances (including flavors, structures, and so on)
- Compiled functions

:fasd-form Operation on *instances*

When an instance is *fasdumped* (put into an object file), it is sent a **:fasd-form** message, which must return a Lisp form that, when evaluated, recreates the equivalent of that instance. This procedure is used because instances are often part of a large data structure, and simply fasdumping all of the instance variables and making a new instance with these same values is unlikely to work. Instances remain **eq**; the **:fasd-form** message is only sent the first time a particular instance is encountered during writing of an object file. If the instance does not accept the **:fasd-form** message, it cannot be fasdumped.

dump-forms-to-file *filename forms-list* &optional *attribute-list* Function

This function writes an object file named *filename* that, in effect, contains the forms in *forms-list*. In other words, when the file is loaded, its effect is the same as evaluating the specified forms. For example:

```
(dump-forms-to-file "foo" '((setq x 1) (setq y 2)))
(load "foo")
x => 1
y => 2
```

The *attribute-list* argument is the file attribute list to be stored in the object file. It is a list of alternating keywords and values corresponding to the -*- line of a source file. Probably most useful is the **:mode** attribute, which identifies whether the file is **:Common-Lisp** or **:Zetalisp**. If you do not specify **:mode** the mode defaults to whichever mode was in effect when the file was written. Another useful keyword in this context is **:package**, whose value in the attribute list specifies the package to be used both in dumping the forms and in loading the file. If no **:package** keyword is present, it defaults to the USER package.

compiler:fasd-symbol-value *filename symbol* Function

This function writes an object file named *filename* that contains the value of *symbol*. When the file is loaded, *symbol* is set to the same value. The *filename* argument is parsed with the same defaults that **load** and **compile-file** use. The file type defaults to xld.

compiler:fasd-font *name* Function

This function writes the *name* font into an object file with the appropriate name (in the SYS:FONTS; directory).

compiler:fasd-file-symbols-properties *filename symbols properties dump-values-p dump-functions-p new-symbol-function* Function

This function provides a way to dump a complex data structure into an object file. The values, the function definitions, and some of the properties of certain symbols are put into the object file in such a way that when the file is loaded, **setq**, **fdefine**, and **putprop** are called on the symbols, appropriately. You can control what happens to symbols discovered in the data structures being fasdumped.

The *filename* argument is the name of the file to be written. It is parsed with the same defaults that **load** and **compile-file** use. The file type defaults to xld.

The *symbols* argument is a list of symbols to be processed. The *properties* argument is a list of properties that are to be fasdumped if they are found on the symbols. The *dump-values-p* and *dump-functions-p* arguments control whether the values and function definitions of the symbols are also dumped.

The *new-symbol-function* argument is a function that is called whenever a new (previously unseen) symbol is found in the structure being dumped. This argument can do nothing, or it can add the symbol to the list to be processed by calling **compiler:fasd-symbol-push**. The value returned by *new-symbol-function* is ignored.

Analyzing Object Files

21.10 All Explorer object files are composed of 16-bit bytes. The first two bytes in the file contain fixed values, which are present so that the system can recognize a proper object file. The next byte is the beginning of the first *group*. A group starts with a byte that specifies an operation. It can be followed by other bytes that are arguments.

Most of the groups in an object file are present to construct objects when the file is loaded. These objects are recorded in the *fasl-table*. Each time an object is constructed, it is assigned the next sequential index in the fasl-table. The indices are used by other groups later in the file to refer back to objects already constructed.

To prevent the fasl-table from becoming too large, the object file can be divided into portions called *whacks*. The fasl-table is cleared out at the beginning of each whack.

The other groups in the object file perform operations such as evaluating a list previously constructed or storing an object into a symbol's function cell or value cell.

If you are having trouble with an object file and want to find out exactly what it does when it is loaded, you can use the **sys:unfasl** function to display a description of each group in the file.

sys:unfasl *input-filename* &optional *output-filename* Function

This function writes a description of the contents of the object file *input-filename* into the output file. The output file type defaults to UNFASL, and the rest of the pathname defaults from *input-filename*.

sys:unfasl-print *input-filename* Function

This function prints on ***standard-output*** a description of the contents of the object file *input-filename*.

Recording Warnings

21.11 The warnings database is not only for compilation. It can record operations for any number of different operations on files or parts of files. Compilation is the only operation in the system that currently uses it.

Each operation for which warnings can be recorded should have a name, preferably in the KEYWORD package. This symbol should have four properties that tell the system how to print out the operation name as various parts of speech. For compilation, the operation name is **:compile** and the properties are defined as follows:

```
(defprop :compile "compilation" name-as-action)
(defprop :compile "compiling" name-as-present-participle)
(defprop :compile "compiled" name-as-past-participle)
(defprop :compile "compiler" name-as-agent)
```

The warnings system considers that these operations are normally performed on files composed of named objects. Each warning is associated with a filename and then with an object within the file. It is also possible to record warnings about objects that are not within any file.

To tell the warnings system that you are starting to process all or part of a file, use the **sys:file-operation-with-warnings** macro.

sys:file-operation-with-warnings Macro
(*generic-pathname operation-name whole-file-p*) {*body-form*}*

The *body* argument is executed within a context set up so that warnings can be recorded for *operation-name* about the file specified by *generic-pathname* (see the *Explorer Input/Output Reference* manual for information about generic pathnames).

In the case of compilation, this procedure is done in **compiler:compile-stream**, which is used to compile a file or editor buffer.

The *whole-file-p* argument should be non-**nil** if the entire contents of the file are to be processed inside the *body-form* (if it finishes); a non-**nil** value implies that any warnings left from previous iterations of this operation on this file should be thrown away on exit. This convention is relevant only to objects that are not found in the file this time, the assumption being that the objects must have been deleted from the file and that their warnings are no longer appropriate.

All three of the special arguments are specified as expressions that are evaluated.

Within the processing of a file for which you are collecting warnings, you must also announce when you are beginning to process an object:

sys:object-operation-with-warnings Macro
(*object-name location-function*) {*body-form*}*

This macro enables warnings to be recorded for a specific object.

The *body-form* argument is executed in a context set up so that warnings are recorded for the *object-name*, which can be a symbol or a list. Object names are compared with **equal**.

In the case of compilation, this macro encompasses the processing of a single function.

The *location-function* argument is either **nil** or a function that the editor uses to find the text of the object. Refer to the source files in ZWEI regarding POSSIBILITIES.

The *object-name* and *location-function* arguments are specified with expressions that are evaluated.

You can enter this macro recursively. If the inner invocation is for the same object as the outer one, it has no effect. Otherwise, warnings recorded in the inner invocation apply to the object specified therein.

Finally, when you detect exceptions, you must make the actual warnings:

sys:record-warning *type severity location-info format-string* &rest *args* Function
sys:record-and-print-warning *type severity location-info format-string* &rest *args* Function

This function records one warning for the object and file currently being processed. The text of the warning is specified by *format-string* and *args*, which are suitable arguments for **format**, but the warning is *not* printed when you call this function. These arguments are used to reprint the warning later.

The **sys:record-and-print-warning** function records a warning and also prints it.

The *type* argument is a symbol that identifies the specific cause of the warning. Types have meaning only as defined by a particular operation, and currently no operations make much use of them. The system defines one type: **sys:premature-warnings-marker**.

The *severity* argument measures how important this warning is and identifies its general causal classification. This argument should be a symbol in the KEYWORD package. Several severities are defined and should be used when appropriate, but no code looks at them:

:implausible — This keyword indicates an event occurred that was not intrinsically wrong but probably due to a mistake of some sort.

:impossible — This warning concerns a situation that cannot have a meaning even if circumstances outside the text being processed are changed.

:probable-error — This keyword indicates an error that can be corrected by a change elsewhere, for example, calling a function with the wrong number of arguments.

:missing-declaration — This keyword is used for warnings about free variables not declared special, and the like. It means that the text is not actually incorrect, but something else that is supposed to accompany it is missing.

:obsolete — This warning indicates something that you should not use any more, but which still works.

:very-obsolete — This indicates something that no longer works.

:maclisp — This keyword indicates something that does not work in MacLisp.

:fatal — This keyword indicates a problem so severe that no sense can be made of the object at all. It indicates that the presence or absence of other warnings is not significant.

:error — There was a Lisp error in processing the object.

:implementation-limit — This keyword indicates something meaningful, but beyond the capacity of the current implementation.

The *location-info* argument is intended to be used to inform the editor of the precise location in the text of the cause of this warning. It is not defined yet, and you should use **nil**.

If a warning is encountered during processing of data that does not actually have a name (such as forms in a source file that are not function definitions), you can record a warning even though you are not inside an invocation of **sys:object-operation-with-warnings**. This warning is known as a *premature warning*, and it is recorded with the next object that is processed; a message is added so that you can tell which warnings were premature.

THE DISASSEMBLER

22

Introduction

22.1 Studying the machine language code (or *macrocode*) produced by the Explorer's compiler can be useful in analyzing errors or in checking for a suspected compiler problem. This section explains how the Explorer instruction set works and how to understand the behavior of code written in this instruction set. Fortunately, the translation between Lisp and this instruction set is not difficult. Once you become familiar with the instruction set, you can easily move between the two representations. This section requires no special knowledge of the Explorer system, although you should be somewhat familiar with computer science in general.

No one examines machine language code by manually interpreting octal numbers. Instead, a program called the disassembler converts the numeric representation of the instruction set into a more readable textual representation. This program is called the disassembler because it does the opposite of what an assembler does. No assembler accepts this input, however, since the Explorer system requires no written assembly language.

The error handler and the Inspector also use the disassembler. If you see code similar to that in Example 2 (later in this section) while using either the error handler or the Inspector, it is disassembled code in the same format as the **disassemble** function uses. Inspecting a compiled code object shows the disassembled code.

The disassemble Function

22.2 The simplest way to invoke the disassembler is with the **disassemble** function.

disassemble *function* &key **:base :verbose :start :end** *[c]* Function

This function prints a humanly readable version of the instructions in *function*.

function — This argument should be a function object, lambda expression, a closure, or a function spec with a function definition. If the function is not compiled, a compiled version is generated, although this version is not used to update the function spec.

:base — This keyword manipulates the print base for all numbers output by this function. The default value for this parameter is the value of ***print-base***.

:verbose — This keyword enables the printing of the numeric representation of the macrocode. The numeric base is octal (8.) unless **:base** is 16.

:start, :end — These keywords allow you to specify that only a portion of the macrocode is to be printed. The **:start** keyword specifies where in the macrocode to start printing, and the **:end** keyword specifies where to stop printing.

The display format is also affected by the ***print-case*** variable. To see a simple example of disassembly, enter the code in Example 1:

Example 1:

```
(defun foo (x)
 (assoc 'key (cdr (get x 'propname))))

(compile 'foo)

(disassemble 'foo)
```

The code you just entered defines the function `foo`, compiles it, and invokes the disassembler to print the textual representation of the result of the compilation. The disassembled code should appear similar to that in Example 2.

Example 2:

```
12 PUSH              FEF|3      ; 'KEY
13 PUSH              ARG|0      ; X
14 PUSH-GET          FEF|4      ; 'PROPNAME
15 (MISC) PUSH CDR
16 RETURN CALL-2     FEF|5      ; #'SYS:ASSOC-EQL
```

Before translating the disassembled code into its component actions, you must first become familiar with some terminology.

The acronym PDL stands for push-down list. A PDL is a stack, a last-in first-out (LIFO) memory. You will see the terms PDL and stack used interchangeably. The Explorer system's architecture is typical of stack machines; that is, it has a stack (PDL) with which most instructions deal. The stack holds the following items:

- Values being computed
- Arguments
- Local variables
- Flow-of-control information (function call frames and the like)

An important use of the stack is to pass arguments to instructions, though not all instructions take their arguments from the stack.

After the `defun` form above is evaluated, the function cell of the symbol `foo` contains a lambda expression. When the function `foo` is compiled, the contents of the function cell are replaced by a compiled function object. The printed representation of the compiled function object for `foo` appears as follows:

```
#<DTP-FUNCTION FOO 11464337>
```

Stated somewhat simply, the compiled function has three parts:

- A header with various fixed-format fields
- A part holding constants and invisible pointers
- The main body, holding the machine language instructions (macrocode)

Compiled functions are also called function entry frames (FEFs).

The first part, the header, is not discussed in this manual. You can look at a representation of the information contained in the header with the **describe** function.

The second part holds various constants referred to by the function; for example, the function `foo` refers to two constants (the symbols `key` and `propname`), so pointers to these symbols are saved. This part of the function also holds invisible pointers to the value cells of all symbols that the function uses as special variables or calls as functions.

The third part holds the macrocode itself.

Now you can read the disassembled code. The first instruction from the preceding example appears as follows:

```
12 PUSH                  FEF|3        ; 'KEY
```

This instruction has several parts. The `12` is the address of this instruction within the compiled function. The disassembler prints out the address of each instruction before it prints out the instruction so that you can interpret branching instructions when you see them (one of these is discussed later in this section). You can control the base of the address printed by using the **:base** keyword.

The `PUSH` moves a piece of data from one place onto the stack.

The next field of the instruction is `FEF|3`. This is an *address* that specifies where the data item comes from. The vertical bar serves to separate the two parts of the address. The part before the vertical bar can be thought of as a *base register*, and the part after the bar can be regarded as an offset from this register (which is zero-origined). `FEF` as a base register means the address of the FEF that you are disassembling, so this address denotes the location four words into the FEF (zero-origined). Thus, this instruction takes the data located four words into the FEF and pushes it onto the PDL.

The instruction is followed by a comment field, which looks like `;'KEY`. This is not a comment written by a person; the disassembler produces these comments to explain what is going on. The semicolon serves to start the comment the way semicolons in Lisp code do. In this case, the body of the comment, `'KEY`, tells you that the address field (`FEF|3`) is addressing a constant (this is what the single quotation mark in `'KEY` means) and that the printed representation of this constant is `KEY`. This comment helps explain what this instruction is actually doing: it is pushing (a pointer to) the `key` symbol onto the stack.

The next instruction is as follows:

```
13 PUSH                  ARG|0        ; X
```

This is much like the previous instruction; the only difference is that a different *base register* is being used in this address. The `ARG` base register is used for addressing your arguments: `ARG|0` means that the data item being addressed is the zeroth argument. Again, the comment field explains that the value of X (which was the zeroth argument) is being pushed onto the stack.

The third instruction is as follows:

```
14 PUSH-GET              FEF|4        ; 'PROPNAME
```

This instruction is much like the previous two; however, it differs in that it pushes the value of a symbol's property and thus uses two arguments. The symbol is received from the top of the stack (the result of the preceding instruction X), and the property name is received with the FEF|4 base addressing described previously.

The fourth instruction is something new:

```
15 (MISC) PUSH CDR
```

The (MISC) form means that this is one of the so-called *miscellaneous* instructions. There are quite a few of these instructions. With some exceptions, each miscellaneous instruction corresponds to a Lisp function and has the same name as this Lisp function. If a Lisp function has a corresponding miscellaneous instruction, then this function is implemented in Explorer microcode.

Miscellaneous instructions only have a destination field; they do not have any address field. The input to the instruction comes from the stack: the top *n* elements on the stack are used as input to the instruction and popped off the stack, where *n* is the number of arguments taken by the function. The result of the function is stored wherever indicated by the destination field. In this case, the function being executed is **cdr**, a Lisp function of one argument. The top value is popped off the stack and used as the argument to **cdr** (generally, the value pushed first is the first argument; the value pushed second is the second argument, and so on). Functions that have optional arguments or that return multiple values cannot become miscellaneous instructions. Functions that return multiple values are almost never miscellaneous instructions.

The fifth and last instruction is as follows:

```
16 RETURN CALL-2        FEF|5        ; #'SYS:ASSOC-EQL
```

This is a CALL macroinstruction, which is discussed in more detail in paragraph 22.4.4, Call Instructions. Here the function at FEF|5 (SYS:ASSOC-EQL) is called with two arguments (CALL-2). Also, the result of this function call is returned (RETURN) as the result of the function foo.

The original Lisp program compiled is as follows:

```
(defun foo (x)
 (assoc 'key (cdr (get x 'propname))))
```

Now, recall the program as a whole and observe what it produces:

```
12 PUSH                 FEF|3        ; 'KEY
13 PUSH                 ARG|0        ; X
14 PUSH-GET             FEF|4        ; 'PROPNAME
15 (MISC) PUSH CDR
16 RETURN CALL-2        FEF|5        ; #'SYS:ASSOC-EQL
```

First, it pushes the key symbol. Then it pushes the value of x. Then it invokes push-get, which pops the value of x, gets the propname symbol from FEF|4, and uses them as arguments, thus performing the equivalent of evaluating the following form:

```
(get x 'propname)
```

The result is pushed on the stack, which now contains the result of the `get` on top and the symbol `key` underneath that. Next, it invokes `cdr` on the result of the `get`, thus performing the equivalent of evaluating the following form:

```
(cdr (get x 'propname))
```

Finally, it calls the function using the result of the `cdr` and the symbol `key`, thus performing the equivalent of evaluating the following form:

```
(assoc 'key (cdr (get x 'propname)))
```

The code produced by the compiler is correct: it produces the same result as the function you defined (`SYS:ASSOC-EQL` is a compiler optimization of `assoc`).

The following four examples show the use of **disassemble** with each of the four keywords. Example 3 demonstrates the use of the **:base** keyword:

Example 3:

```
(disassemble 'foo :base 16.)
```

The disassembled code appears as follows:

```
 C PUSH                     FEF|3        ; 'KEY
 D PUSH                     ARG|0        ; X
 E PUSH-GET                 FEF|4        ; 'PROPNAME
 F (MISC) PUSH CDR
10 RETURN CALL-2            FEF|5        #'SYS:ASSOC-EQL
```

Example 4 shows the use of the **:verbose** keyword, which is set to t:

Example 4:

```
(disassemble 'foo :verbose t)
```

The disassembled code appears as follows. Note that a macroinstruction word is 16 bits long and is displayed as six octal digits:

```
12 050003 PUSH              FEF|3        ; 'KEY
13 050600 PUSH              ARG|0        ; X
14 057004 PUSH-GET          FEF|4        ; 'PROPNAME
15 041003 (MISC) PUSH CDR
16 112005 RETURN CALL-2     FEF|5        ; #'SYS:ASSOC-EQL
```

Example 5 shows the use of both **:base** and **:verbose**:

Example 5:

```
(disassemble 'foo :verbose t :base 16.)
```

The disassembled code appears as follows:

```
 C 5003 PUSH                FEF|3        ; 'KEY
 D 5180 PUSH                ARG|0        ; X
 E 5E04 PUSH-GET            FEF|4        ; 'PROPNAME
 F 4203 (MISC) PUSH CDR
10 9405 RETURN CALL-2       FEF|5        ; #'SYS:ASSOC-EQL
```

Notice that `:base` affects both the instruction address and the numeric representation.

Example 6 shows the use of the two previous keywords and the **:start** and **:end** keywords:

Example 6:

```
(disassemble 'foo :verbose t :base 16. :start 12. :end 14.)
```

The disassembled code appears as follows:

```
C 5003 PUSH             FEF|3       ; 'KEY
D 5180 PUSH             ARG|0       ; X
```

In summary, four kinds of instructions have been presented thus far: the PUSH instruction, which takes an address for an operand and places the operand onto the stack; the PUSH-GET instruction, which uses two operands and thus must pop one of them off the stack; the (MISC) instruction, one of the members of the large set of miscellaneous instructions, which take only a destination and implicitly receive their input from the stack; and, finally, the CALL instruction, which returns as the result of the function the single value returned by the SYS:ASSOC-EQL function call. Moreover, two forms of addressing (FEF addressing and ARG addressing) have been discussed.

An Advanced Example

22.3 Example 7 is a more complex function than those discussed in the previous paragraph demonstrating local variables, function calling, conditional branching, and some other new instructions.

Example 7:

```
(defun bar (y)
 (let ((z (car y)))
   (cond ((atom z)
          (setq z (cdr y))
          (foo y))
         (t nil))))

(disassemble 'foo)
```

```
 8 PUSH-CAR            ARG|0       ; Y
 9 POP                 LOCAL|0     ; Z
10 BR-NOT-ATOM  15
11 PUSH-CDR            ARG|0       ; Y
12 POP                 LOCAL|0     ; Z
13 PUSH                ARG|0       ; Y
14 RETURN CALL-1       FEF|3       ; #'FOO
15 (AUX) RETURN-NIL
```

The first instruction here is a PUSH-CAR instruction that has the same format as PUSH: with a destination and an address. The PUSH-CAR instruction reads the data addressed by the address, takes the car of it, and pushes the result onto the stack. In our example, the first instruction addresses the zeroth argument, so it computes the following:

```
(car y)
```

Then it pushes the result onto the stack (the destination).

The next instruction is something new: the POP instruction. It has an address field, but it uses this field as a destination rather than as a source. The POP instruction pops the top value off the stack and stores this value into the address specified by the address field. In our example, the value on the top of the stack is popped off and stored into address LOCAL|0. This is a new form of address, which means the zeroth local variable.

The ordering of the local variables is chosen by the compiler, so it is not fully predictable, although it tends to be by order of appearance in the code.

Fortunately, you seldom have to look at these numbers because the comment field explains what is going on. In this case, the variable being addressed is z. Thus, this instruction pops the top value on the stack into the variable z. The first two instructions work together to take the car of y and store it into z, which is indeed the first action that the function bar should take.

If you have two local variables with the same name, as happens with lexical shadowing, then the comment field does not distinguish between the two. You can distinguish between two local variables with the same name by looking at the number in the address.

Every instruction that moves or produces a data item sets the indicator bits from this data item so that subsequent instructions can test them. As a result, the POP instruction allows someone to test the indicators set up by the value that was moved, namely the value of z .

The next instruction is a conditional branch, which changes the flow of control on the basis of the values in the indicator bits. The instruction is BR-NOT-ATOM 15, which means "Branch, if the quantity was not an atom, to location 15; otherwise, proceed with execution". If z was not an atom, execution branches to location 15 and proceeds from there. Location 15 contains a RETURN-NIL instruction, which causes the function to return **nil**.

If z is an atom, the program continues, and the PUSH-CDR instruction is executed next. This instruction resembles PUSH-CAR except that it takes the cdr. It pushes onto the stack the value of the following:

```
(cdr y)
```

The next instruction pops this value off into the z variable.

The last two instructions provide an example of how function calling is compiled. The following demonstrates how it works in our example:

```
13 PUSH              ARG|0      ; Y
14 RETURN CALL-1     FEF|3      ; #'FOO
```

The form being compiled here is the following:

```
(foo y)
```

Thus, you apply the function in the function cell of the symbol foo and pass it one argument: the value of y. Simple function calling works in the following two steps. First, all the arguments being passed to the function are pushed onto the stack. Second, the CALL-1 instruction specifies the function object being applied to the arguments. This instruction creates a new stack frame on the stack and stores the function object there. There are variations to the CALL-1 macroinstruction, namely CALL-0 to CALL-6 and CALL-N. The number following the CALL- indicates the number of arguments that the function object expects. CALL-N is a generic calling macroinstruction. After N arguments are pushed, then the actual number of the arguments, N, is pushed and the instruction CALL-N is invoked. For example, if a function is called with seven arguments, the seven arguments are pushed, then the number 7 is pushed, and finally the CALL-N macroinstruction is invoked specifying the function object to call.

When the function returns, the destination field of the CALL-1 instruction determines what happens to the returned value (RETURN). When the function actually returns, its result is stored into this destination. A destination of RETURN causes the result of the function call also to be the value returned from this function.

Thus, in the two-instruction sequence above, the first instruction is a PUSH; the value to push is located at ARG|0, which, as the comment indicates, is the value Y. Next, the CALL-1 instruction is executed; the function object it specifies is at FEF|3, which, as the comment indicates, is the contents of the function cell of FOO (the FEF contains an invisible pointer to this function cell). The destination field of the CALL-1 is RETURN, indicating that the result of this function call is also the returned value for this function.

The following is another example to illustrate function calling. This Lisp function calls one function on the results of another function:

Example 8:

```
(defun a (x y)
 (b (c x y) y))
```

The disassembled code is as follows:

```
10 PUSH               ARG|0     ; X
11 PUSH               ARG|1     ; Y
12 PUSH CALL-2        FEF|3     ; #'C
13 PUSH               ARG|1     ; Y
14 RETURN CALL-2      FEF|4     ; #'B
```

The first two macroinstructions push the arguments x and y for the function c. Next, the function c is called with the CALL-2 macroinstruction. Notice that the destination of this call is PUSH, indicating that the result of the function call is to be pushed onto the stack. Then the argument y is pushed for the function b (the other argument is the result of the CALL-2). Finally, the function b is called with two arguments on the stack and with a destination of RETURN, indicating that the value returned by b is also to be returned as the result of function a.

Macroinstruction Classes

22.4 In general, macroinstructions fall into seven classes:

- Main operations (or *main ops*)
- Short branches
- Immediate operations
- Call instructions
- Miscellaneous operations (or *misc ops*)
- Auxiliary operations (or *aux ops*)
- Module operations (or *module ops*)

The instruction set is defined by the file SYS:UCODE;DEFOP.LISP, which uses several special forms that are defined in the file SYS:COMPILER;TARGET. What follows is a description of the various instruction formats. See the *Explorer System Software Design Notes* for more detailed information about individual macroinstructions.

Main Operation Instructions

22.4.1 Main op instructions have an operand address as part of the instruction (like PUSH, described earlier) and can take additional operands from the stack (like PUSH-GET, described earlier). The result of the main op is implied by the operation; for example, PUSH places the result on the top of the stack. Table 22-1 lists the 54 main ops.

Table 22-1 **Main Operation Instructions**

ARRAYP	POP	SETE-1+
BIND-CURRENT	PUSH	SETE-1-
BIND-NIL	PUSH-AR-1	STRINGP
BIND-POP	PUSH-CADDR	TEST
BIND-T	PUSH-CADR	TEST-CAAR
EQ	PUSH-CAR	TEST-CADR
EQL	PUSH-CDDR	TEST-CAR
EQUAL	PUSH-CDR	TEST-CDDR
EQUALP	PUSH-CDR-STORE-CAR-IF-CONS	TEST-CDR
FIXNUMP	PUSH-CONS	TEST-MEMQ
INTEGERP	PUSH-GET	1+
LISTP	PUSH-LOC	1-
LOGAND	RETURN	+
LOGXOR	SET-NIL	-
MINUSP	SET-T	*
MOVEM	SET-ZERO	=
NUMBERP	SETE-CDDR	>
PLUSP	SETE-CDR	<

The **PUSH** instruction has already been discussed. **PUSH-CADDR, PUSH-CADR, PUSH-CAR, PUSH-CDDR,** and **PUSH-CDR** are similar to **PUSH** except that the **CADDR, CADR, CAR, CDDR,** or **CDR**, respectively, is taken from the operand addressed as part of the instruction and returned on top of the stack.

PUSH-CONS and **PUSH-AR-1** are similar to **PUSH-GET** in that an additional operand is taken from the stack. **PUSH-CONS** pops the car from the stack, receives the cdr from the addressed operand, and pushes the resultant cons on the stack. **PUSH-AR-1** is used to access elements within a one-dimensional array. The index for the array element to be accessed is taken from the stack, and the array itself is received from the addressed operand. The resultant array element is pushed on the stack. The stack level does not change with **PUSH-GET, PUSH-CONS,** or **PUSH-AR-1.**

PUSH-CDR-STORE-CAR-IF-CONS is used primarily to implement the Lisp form **dolist.** Consider the following example:

Example 9:

```
(defun p (list)
   (dolist (x list)
      (print x)))
```

The disassembled code is as follows:

```
 8 PUSH                           ARG|0      ; LIST
 9 PUSH-CDR-STORE-CAR-IF-CONS     LOCAL|0    ; X
10 BR-NULL                        14
11 PUSH                           LOCAL|0    ; X
12 TEST CALL-1                    FEF|3      ; #'PRINT
13 BR                             9
14 (AUX) RETURN-NIL
```

In this example, instruction 8 pushes onto the stack the argument for the function (`list`). Instruction 9 is the new instruction PUSH-CDR-STORE-CAR-IF-CONS. This instruction takes its first argument from the stack (`list` in this case) and its second argument from the specified address (LOCAL|0). This instruction pops `list`, and if `list` is a cons, then it pushes the cdr of `list` to replace it and stores the car of `list` in LOCAL|0. If `list` is not a cons (which is eventually the case because successive cars of `list` are removed), then nothing is pushed or stored and the symbol **nil** is left in the indicators. Instruction 10 checks for this last case and branches to instruction 14 when the end of `list` is reached. Instruction 11 immediately retrieves the car of `list` stored in LOCAL|0 by PUSH-CDR-STORE-CAR-IF-CONS. This car is used as a functional argument for `print`, which is called in instruction 12 with the CALL-1 instruction. The destination type TEST in the CALL-1 instruction implies that the result of the `print` functional call is not used (it is not pushed on the stack or returned as the result of this function). Rather, only the indicators are set in case the code desires to test it. Instruction 13 is an unconditional branch to instruction 9. Instruction 14 (an aux op) is executed when all of `list` has been traversed and returns **nil** as the result of this function.

The **TEST** instructions (**TEST**, **TEST-CAAR**, **TEST-CADR**, **TEST-CAR**, **TEST-CDDR**, and **TEST-CDR**) are exactly like their associated **PUSH** instructions except that these instructions have the same destination type as the CALL-1 instruction in the previous example. That is, the result of these instructions is not pushed on the stack but is used only to set the indicators for immediate testing (the conditional branch instructions).

TEST-MEMQ is used to set the indicators based on the result of a **MEMQ** on the list received from the addressed operand and the element popped off the stack. This instruction corresponds to the Common Lisp function **member** with a **:test** argument of **#'eq**.

The two instructions **1+** and **1–** are examples of frequently used one-argument functions. These instructions take their argument from the specified address, increment or decrement this argument, and leave the result of top of the stack.

Five main op instructions implement heavily used two-argument functions: **+**, **–**, *****, **LOGAND**, and **LOGXOR**. These instructions take their first argument from the top of the stack (popping it off) and their second argument from the specified address. Then, they push their result on the stack. Thus, the stack level does not change because of these instructions.

The following small function shows some of the previously mentioned main ops:

Example 10:

```
(defun foo (x y)
 (setq x (logxor y (* x 5.))))
```

The disassembled code is as follows:

```
 6 PUSH-NUMBER        5
 7 *                  ARG|0      ; X
 8 LOGXOR             ARG|1      ; Y
 9 MOVEM              ARG|0      ; X
10 RETURN             PDL-POP
```

Instructions 7 and 8 use two of the new main op instructions: the `*` and `LOGXOR` instructions. Instruction 9 uses the `MOVEM` instruction; the compiler wants to use the top value of the stack to store it into the value of `x`, but it does not want to pop it off the stack because it plans to return it from the function. Instruction 10 then returns the argument addressed by its operand, `PDL-POP`.

Another 15 main op instructions implement some commonly used predicates: **ARRAYP, EQ, EQL, EQUAL, EQUALP, FIXNUMP, INTEGERP, LISTP, MINUSP, NUMBERP, PLUSP, STRINGP, =, >,** and **<**. The arguments come from the top of the stack (if two arguments are needed) and the specified address; the stack is popped, the predicate is applied to the two objects, and the result is left in the indicators so that a branch instruction can test it and then branch, according to the result of the comparison. These instructions remove the top item on the stack and do not put anything back.

Next, four main op instructions read, modify, and write a quantity in ways that are common in Lisp code. These instructions are **SETE-CDR, SETE-CDDR, SETE-1+,** and **SETE-1–**. The **SETE-** means to set the addressed value to the result of applying the specified one-argument function to the present value. For example, **SETE-CDR** means to read the value addressed, apply **cdr** to it, and store the result back in the specified address. This instruction is used when compiling the following form, which commonly occurs in loops:

```
(setq x (cdr x))
```

Four instructions bind special variables (that is, they save the current value of the variable), but they differ in what they bind the variable with. The first, **BIND-NIL**, binds the cell addressed by the address field to **nil**; the second, **BIND-POP**, binds the cell to an object popped off the stack. The third instruction, **BIND-T**, binds the cell addressed by the address field to **T**. The fourth instruction, **BIND-CURRENT**, binds the cell to its current value (that is, it leaves its value as it was).

Three instructions store common values into addressed cells. **SET-NIL** stores **nil** into the cell specified by the address field, **SET-ZERO** stores **0**, and **SET-T** stores **T**. These instructions do not use the stack at all.

Finally, the **PUSH-LOC** instruction creates a locative pointer to the cell referenced by the specified address and pushes it onto the stack. This instruction is used in compiling the following form:

```
(variable-location z)
```

In this example, `z` is an argument or a local variable rather than a special variable.

Example 11 uses some of these instructions to show what they look like:

Example 11:

```
(defvar *foo*)

(defvar *bar*)

(defun weird (x y)
   (cond ((= x y)
          (let ((*foo* nil) (*bar* 5))
            (setq x (cdr x)))
          nil)
         (t
          (setq x nil)
          (caar (variable-location y)))))
```

The disassembled code appears as follows:

```
16 PUSH              ARG|0       ; X
17 =                 ARG|1       ; Y
18 BR-NULL  24
19 BIND-NIL          FEF|3       ; *FOO*
20 PUSH-NUMBER       5
21 BIND-POP          FEF|4       ; *BAR*
22 SETE-CDR          ARG|0       ; X
23 (AUX) RETURN-NIL
24 SET-NIL           ARG|0       ; X
25 PUSH-LOC          ARG|1       ; Y
26 (MISC) PUSH CAAR
27 RETURN            PDL-POP
```

Instruction 17 is an `=` instruction; it numerically compares the top of the stack, `x`, with the addressed quantity, `y`. The `x` is popped off the stack, and the indicators are set to the result of the equality test. Instruction 18 checks the indicators, branching to instruction 24 if the result of `=` is **nil**; that is, the machine branches to 24 if the two values are not equal. Instruction 19 binds `*foo*` to **nil**. Instructions 20 and 21 bind `*bar*` to 5. Instruction 22 demonstrates the use of `SETE-CDR` to compile the following form:

```
(setq x (cdr x))
```

Instruction 24 demonstrates the use of `SET-NIL` to compile the form:

```
(setq x nil)
```

Instruction 25 demonstrates the use of `PUSH-LOC` to compile the form:

```
(variable-location y)
```

Short Branch Instructions

22.4.2 Short branch instructions are the branch instructions that contain a branch address rather than a general base-and-offset address. These instructions have neither addresses nor destinations of the usual sort. Instead, they have branch addresses and indicate where to branch if the branch is going to happen. Branching instructions differ in the conditions under which they branch and whether they pop the stack. Branch addresses are stored internally as self-relative addresses to make Explorer code relocatable, but the disassembler performs the addition for you and prints out FEF-relative addresses so that you can easily see where the branch is going. If the relative address into the FEF is too large for encoding within this instruction, the long branch instruction is used. Long branch instructions are discussed in paragraph 22.4.6.3, Long Branches.

The branch instructions discussed so far decide whether to branch on the basis of the *nil indicator*, that is, whether the last value dealt with was **nil** or non-**nil**. **BR-NIL** branches if it was **nil**, and **BR-NOT-NIL** branches if it was not **nil**. Three other pairs of branch instructions use the **atom**, **zerop**, and **symbolp** predicates, respectively. **BR-ATOM** branches if the value was an atom (that is, if it was anything besides a cons), and **BR-NOT-ATOM** branches if the value was not an atom (that is, if it was a cons). **BR-ZEROP** branches if the value is zero, and **BR-NOT-ZEROP** branches if the value is not zero. **BR-SYMBOLP** branches if the value is a symbol, and **BR-NOT-SYMBOLP** branches if the value was not a symbol.

The **BR** instruction is an unconditional branch (it always branches).

Table 22-2 lists all of the short branch instructions.

Table 22-2 **Short Branch Instructions**

BR	**BR-NOT-ATOM**	**BR-NOT-ZEROP**
BR-ATOM	**BR-NOT-NIL**	**BR-SYMBOLP**
BR-NIL	**BR-NOT-NIL-ELSE-POP**	**BR-ZEROP**
BR-NIL-ELSE-POP	**BR-NOT-SYMBOLP**	

None of the above branching instructions deal with the stack. The two instructions called **BR-NIL-ELSE-POP** and **BR-NOT-NIL-ELSE-POP** are the same as **BR-NIL** and **BR-NOT-NIL** except that if the branch is not performed, the top value on the stack is popped off the stack. These are used for compiling **and** and **or** special forms.

Immediate Operation Instructions

22.4.3 Immediate operation instructions use the lower nine bits of the instruction word (immediate operand) in special ways.

No Lisp functions directly correspond to the immediate operations, which are used strictly by the compiler.

A previous example has already shown the use of **PUSH-NUMBER**, which pushes the number in the lower nine bits of the instruction onto the stack. **PUSH-NEG-NUMBER** negates the number before pushing it. **PUSH-LONG-FEF** uses the immediate operand as an index into the current FEF and pushes the contents that reside at that location. This instruction is used only when the regular FEF base addressing can no longer reach the desired location.

The **=-IMMED**, **>-IMMED**, **<-IMMED**, and **EQ-IMMED** instructions all perform the condition on the immediate operand and the value popped off the stack. Nothing is pushed as a result of these instructions, but the indicators are set.

ADD-IMMED adds the immediate operand to the value on top of the stack. The result is left on top of the stack.

LDB-IMMED uses the immediate operand to describe the bits to extract from the operand on top of the stack, and the resultant value is pushed on top of the stack. This immediate value is broken up into a five-bit position value (immediate operand bits 4 through 8) and a four-bit length value (immediate operand bits 0 through 3). Note that the format for this immediate value is different from that for a byte specifier.

The **DISPATCH** instruction is similar to a **BR** instruction except that it allows a multiway transfer of control. The compiler uses the **DISPATCH** instruction to initialize optional arguments to a function. Consider the following example:

Example 12:

```
(defun f (&optional (a 1) (b 2) (c 3))
   (list a b c))
```

The disassembled code appears as follows:

```
20 DISPATCH          FEF|4     ; [0->21;1->23;2->25;3->27;ELSE->27]
21 PUSH-NUMBER       1
22 POP               ARG|0     ; A
23 PUSH-NUMBER       2
24 POP               ARG|1     ; B
25 PUSH-NUMBER       3
26 POP               ARG|2     ; C
27 PUSH              ARG|0     ; A
28 PUSH              ARG|1     ; B
29 PUSH              ARG|2     ; C
30 RETURN CALL-3     FEF|3     ; #'LIST
```

The **DISPATCH** instruction uses the number of &optional parameters supplied (this argument is supplied on top of the stack by the function-calling microcode) as the index into the table at `FEF|4`. The comment for the **DISPATCH** instruction indicates where control will be transferred for the various number of &optional parameters supplied. For example, if no optional parameters are supplied, control is transferred to the instruction at 21 where `a`, `b`, and `c` are initialized.

The **SELECT** instruction uses the immediate operand as a nine-bit FEF offset, which addresses a select table. The value on top of the stack is looked up in this table. The offset of the selected slot within the select table is then used as an index into the dispatch table to allow a multiway transfer of control. The **SELECT** instruction is used by the compiler optimization of Lisp forms such as **selectq** and **case**.

The **LEXICAL-UNSHARE** and **LOCATE-LEXICAL-ENVIRONMENT** instructions use only the immediate operand and are used by the compiler in the implementation of lexical closures.

Table 22-3 lists all the immediate operation instructions.

Table 22-3

Immediate Operation Instructions

ADD-IMMED
DISPATCH
EQ-IMMED
LDB-IMMED
LEXICAL-UNSHARE
LOCATE-LEXICAL-ENVIRONMENT
PUSH-LONG-FEF
PUSH-NEG-NUMBER
PUSH-NUMBER
SELECT
=-IMMED
>-IMMED
<-IMMED

Call Instructions

22.4.4 The simple call instructions consist of **CALL-0** through **CALL-6** and **CALL-N**. The base and offset field is used to specify the function to be called. The function arguments are pushed on the stack before executing these instructions. If the number of arguments is more than six, then the **CALL-N** instruction is used, and the number of arguments is the last item pushed on the stack.

Complex function calling (such as returning multiple values) is described in paragraph 22.4.6.1, Simple Aux Ops.

Table 22-4 lists all the simple call instructions.

Table 22-4

Call Instructions
CALL-0
CALL-1
CALL-2
CALL-3
CALL-4
CALL-5
CALL-6
CALL-N

Miscellaneous Operation Instructions

22.4.5 Miscellaneous operation instructions (or misc ops) take their arguments from the stack and produce a resultant value that sets the indicators and is optionally pushed on the stack.

Table 22-5 lists all the misc ops.

Table 22-5 Miscellaneous Operation Instructions

ABS
AP-LEADER
AP-1
AP-1-FORCE
AP-2
AP-3
ARRAY-ACTIVE-LENGTH
ARRAY-DIMENSION
ARRAY-HAS-FILL-POINTER-P
ARRAY-HAS-LEADER-P
ARRAY-LEADER
ARRAY-LEADER-LENGTH
ARRAY-LENGTH
ARRAYP
ARRAY-PUSH
ARRAY-RANK
ARRAY-TOTAL-SIZE
AR-1
AR-1-FORCE
AR-2
AR-2-REVERSE
AR-3
ASH
ASSQ
AS-1
AS-1-FORCE
AS-2
AS-2-REVERSE
CAAAR
CAADAR
CAADDR
CAADR
CAAR
CADAAR
CADADR
CADAR
CADDAR
CADDDR
CADDR
CADR
CAR
CARCDR
CAR-SAFE
CDAAAR
CDAADR
CDAAR
CDADAR
CDADDR
CDADR
CDAR
CDDAAR
CDDADR
CDDAR
CDDDAR
CDDDDR
CDDDR

NOTE: The functions that are microcoded are subject to change. Functions may be added to or deleted from this list without notice.

Table 22-5 Miscellaneous Operation Instructions (Continued)

AS-3
ATOM
BIGNUM-TO-ARRAY
BIND
BITBLT
BIT-VECTOR-P
BOUNDP
CAAAAR
CAAADR
COMMON-LISP-AR-1-FORCE
COMMON-LISP-AR-2
COMMON-LISP-AR-3
COMMON-LISP-ELT
COMMON-LISP-LISTP
COMPLEXP
CONS
CONS-IN-AREA
CONSP-OR-POP
COPY-ARRAY-CONTENTS
COPY-ARRAY-CONTENTS-AND-LEADER
COPY-ARRAY-PORTION
DEPOSIT-FIELD
DOUBLE-FLOAT
DOUBLE-FLOATP
DPB
ELT
ENDP
EQ
EQL
EQ-T
EQUAL
EQUALP
FBOUNDP
FIND-POSITION-IN-LIST
FIX
FIXNUMP
FIXP
FLOAT-EXPONENT
FLOAT-FRACTION
FLOATP
FLOOR-1
FLOOR-2
FSYMEVAL
FUNCTION-CELL-LOCATION
GCD
GETL
GET-LEXICAL-VALUE-CELL
GET-LOCATION-OR-NIL
GET-PNAME
G-L-P
HAULONG
INT-CHAR
INTERNAL-CHAR-EQUAL
CDDR
CDR
CDR-SAFE
CEILING-1
CEILING-2
CHARACTERP
CHAR-INT
CLOSURE
COMMON-LISP-AR-1
LOGIOR
LSH
MAKE-EPHEMERAL-LEXICAL-CLOSURE
MAKE-LEXICAL-CLOSURE
MASK-FIELD
MAX
MEMQ
MIN
MINUS
MINUSP
NAMED-STRUCTURE-P
NCONS
NCONS-IN-AREA
NLISTP
NOT
NOT-INDICATORS
NSYMBOLP
NTH
NTHCDR
NUMBERP
PDL-WORD
PLUSP
PREDICATE
PROPERTY-CELL-LOCATION
QUOTIENT
RATIONALP
RATIOP
REMAINDER
ROT
ROUND-1
ROUND-2
RPLACA
RPLACD
SCALE-FLOAT
SET
SET-ARRAY-LEADER
SET-AR-1
SET-AR-1-FORCE
SET-AR-2
SET-AR-3
SET-CAR
SET-CDR
SETELT
SHRINK-PDL-SAVE-TOP

NOTE: The functions that are microcoded are subject to change. Functions may be added to or deleted from this list without notice.

Table 22-5 Miscellaneous Operation Instructions (Continued)

INTERNAL-FLOAT
INTERNAL-GET-2
INTERNAL-GET-3
LAST
LDB
LENGTH
LENGTH-GREATERP
LIST-OR-ARRAY
LISTP
LOAD-FROM-HIGHER-CONTEXT
LOCATE-IN-HIGHER-CONTEXT
LOCATE-IN-INSTANCE
SYMBOL-FUNCTION
SYMBOL-NAME
SYMBOLP
SYMBOL-PACKAGE
SYMBOL-VALUE
SYMEVAL
TIME-IN-60THS
TRUNCATE-1
TRUNCATE-2
TYPEP-STRUCTURE-OR-FLAVOR
UNBIND-TO-INDEX-MOVE
VALUE-CELL-LOCATION
VECTORP
VECTOR-PUSH
ZEROP
%ADD-INTERRUPT
%ADD-PAGE-DEVICE
%ALLOCATE-AND-INITIALIZE
%ALLOCATE-AND-INITIALIZE-ARRAY
%ALLOCATE-AND-INITIALIZE-INSTANCE
%AREA-NUMBER
%BLT
%BLT-FROM-PHYSICAL
%BLT-TO-PHYSICAL
%BLT-TYPED
%CHANGE-PAGE-STATUS
%COMPUTE-PAGE-HASH
%DATA-TYPE
%DELETE-PHYSICAL-PAGE
%DIV
%EXTERNAL-VALUE-CELL
%FINDCORE
%FIND-STRUCTURE-HEADER
%FIND-STRUCTURE-LEADER
%FIXNUM-MICROSECOND-TIME
%FUNCTION-INSIDE-SELF
%GC-SCAV-RESET
%GET-SELF-MAPPING-TABLE
%INSTANCE-LOC
%INSTANCE-REF
SIMPLE-ARRAY-P
SIMPLE-BIT-VECTOR-P
SIMPLE-STRING-P
SIMPLE-VECTOR-P
SINGLE-FLOATP
SMALL-FLOAT
SMALL-FLOATP
SPECIAL-PDL-INDEX
STACK-GROUP-RESUME
STACK-GROUP-RETURN
STORE-ARRAY-LEADER
STRINGP
%MICROSECOND-TIME
%NUBUS-READ
%NUBUS-READ-8B
%NUBUS-READ-8B-CAREFUL
%NUBUS-READ-16B
%NUBUS-WRITE
%NUBUS-WRITE-8B
%NUBUS-WRITE-16B
%NUBUS-WRITE-32B
%P-CDR-CODE
%P-CONTENTS-AS-LOCATIVE
%P-CONTENTS-AS-LOCATIVE-OFFSET
%P-DATA-TYPE
%P-DEPOSIT-FIELD
%P-DEPOSIT-FIELD-OFFSET
%P-DPB
%P-DPB-OFFSET
%P-LDB
%P-LDB-OFFSET
%P-MASK-FIELD
%P-MASK-FIELD-OFFSET
%P-POINTER
%P-STORE-CDR-CODE
%P-STORE-CONTENTS
%P-STORE-CONTENTS-OFFSET
%P-STORE-DATA-TYPE
%P-STORE-POINTER
%P-STORE-TAG-AND-POINTER
%PAGE-IN
%PAGE-STATUS
%PAGE-TRACE
%PHYSICAL-ADDRESS
%POINTER
%POINTER-DIFFERENCE
%RATIO-CONS
%RECORD-EVENT
%REGION-NUMBER
%STACK-FRAME-POINTER
%STORE-CONDITIONAL
%STRING-EQUAL

NOTE: The functions that are microcoded are subject to change. Functions may be added to or deleted from this list without notice.

Table 22-5 Miscellaneous Operation Instructions (Continued)

%IO
%LOGDPB
%LOGLDB
%MAKE-EXPLICIT-STACK-LIST
%MAKE-EXPLICIT-STACK-LIST*
%MAKE-LIST
%MAKE-POINTER
%MAKE-POINTER-OFFSET
%MAKE-REGION
%MAKE-STACK-LIST
%MAKE-STACK-LIST*
%STRING-SEARCH-CHAR
%STRING-WIDTH
%STRUCTURE-BOXED-SIZE
%STRUCTURE-TOTAL-SIZE
%SXHASH-STRING
%TEST&SET-68K
%WRITE-INTERNAL-PROCESSOR-MEMORIES
*BOOLE
=
<
>

NOTE: The functions that are microcoded are subject to change. Functions may be added to or deleted from this list without notice.

Most misc ops correspond to Lisp functions, including the subprimitives, although some of these functions are very low level internals that may not be documented anywhere (do not expect to understand all of them). The compiler automatically uses these functions to speed up processing when it can.

The only definitive way to tell if your code is using a microcoded function is to compile some code that uses it and then look at the results, since the compiler occasionally converts a documented function with one name into an undocumented subprimitive.

Auxiliary Operation Instructions

22.4.6 Auxiliary operation instructions (or aux ops) are similar to misc ops except that they do not produce any resultant value, although some of them set the indicators. Aux ops can be divided into four groups: simple aux ops, complex call, long branches, and aux ops with a count field.

Simple Aux Ops

22.4.6.1 Table 22-6 lists all the simple aux ops.

Table 22-6 **Simple Auxiliary Operation Instructions**

BREAKPOINT
CRASH
EXCHANGE
HALT
LEXICAL-UNSHARE-ALL
POPJ
POP-M-UNDER-N
RETURN-NOT-INDS
RETURN-PRED
STORE-IN-HIGHER-CONTEXT
UNBIND-TO-INDEX
%CLOSE-CATCH-RETURN
%CREATE-PHYSICAL-PAGE
%DISABLE-NUPI-LOCKING
%DISK-RESTORE
%ENABLE-NUPI-LOCKING
%GC-CONS-WORK
%GC-FLIP
%GC-FREE-REGION
%GC-SCAVENGE
%OPEN-CATCH
%OPEN-CATCH-MULTIPLE-VALUE
%OPEN-CATCH-MV-LIST
%SET-SELF-MAPPING-TABLE
%SPREAD
%THROW
%THROW-N
%USING-BINDING-INSTANCES
*UNWIND-STACK

NOTE: The functions that are microcoded are subject to change. Functions may be added to or deleted from this list without notice.

Complex Call **22.4.6.2** When you call a function and expect more than one value returned, a slightly different kind of function calling is used. Example 13 uses **multiple-value-setq** to receive two values from a function call:

Example 13:

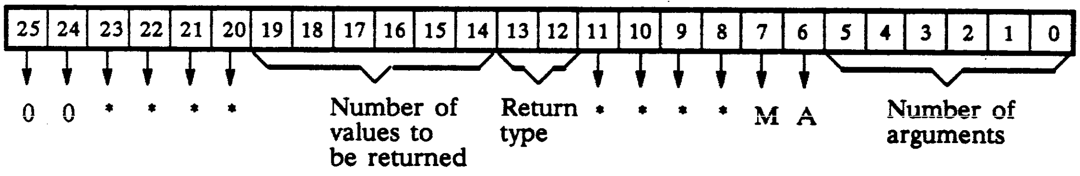
```
(defun foo (x)
 (let (y z)
   (multiple-value-setq (y z)
     (bar 3))
   (* x y z)))
```

The disassembled code appears as follows:

```
10 PUSH-NUMBER          3
11 PUSH                 FEF|3       ; '36865
12 PUSH                 FEF|4       ; #'BAR
13 (AUX) COMPLEX-CALL-TO-PUSH
14 POP                  LOCAL|1     ; Z
15 MOVEM                LOCAL|0     ; Y
16 *                    ARG|0       ; X
17 *                    LOCAL|1     ; Z
18 RETURN               PDL-POP
```

An `(AUX) COMPLEX-CALL-TO-PUSH` instruction is used instead of a `CALL` instruction. The destination field of `(AUX) COMPLEX-CALL-TO-PUSH` is `PUSH`, meaning that the result is pushed on the stack. The `(AUX) COMPLEX-CALL-TO-PUSH` instruction takes two arguments, which it finds on the stack. It pops both of them. The first one is the function object to be applied; the second is the *call-info word*. The call-info word is an integer containing the fields shown in Figure 22-1.

Figure 22-1 Call-Info Word



Return type

0 = Only one result needed
1 = Multiple-value return
2 = Multiple-value list return
3 = Return all values with count on stack

M = Mapping table provided
A = APPLY (spread last argument)
* = Fields that are 0 in the call-info word given to the complex call instruction but are filled in later by the microcode for its own purposes

2287642

The rest of the call proceeds as usual, and when the call returns, the returned values are left on the stack. The number of objects left on the stack is encoded in the call-info word. In this example, the two values returned are left on the stack, and they are immediately popped off into `z` and `y`.

The **multiple-value-bind** form works similarly, as in Example 14:

Example 14:

```
(defun foo (x)
 (multiple-value-bind (y *foo* z)
     (bar 3)
   (+ x y z)))
```

The disassembled code appears as follows:

```
12 PUSH-NUMBER                     3
13 PUSH                            FEF|4          ; '53249
14 PUSH                            FEF|5          ; #'BAR
15 (AUX) COMPLEX-CALL-TO-PUSH
16 POP                             LOCAL|1        ; Z
17 BIND-POP                        FEF|3          ; *FOO*
18 MOVEM                           LOCAL|0        ; Y
19 *                               ARG|0          ; X
20 *                               LOCAL|1        ; Z
21 RETURN                          PDL-POP
```

The `(AUX) COMPLEX-CALL-TO-PUSH` instruction is again used, leaving the results on the stack; these results are used to bind the variables.

Calls performed with **multiple-value-list** also work with the `(AUX) COMPLEX-CALL-TO-PUSH` instruction. Note that the call-info word has a multiple-value list return encoded within it. When the function returns, the list of values is left on the top of the stack. The following is an example of the use of this instruction:

Example 15:

```
(defun foo (x y)
 (multiple-value-list (foo y x)))
```

The disassembled code appears as follows:

```
10 PUSH                            ARG|1          ; Y
11 PUSH                            ARG|0          ; X
12 PUSH                            FEF|3          ; '8194
13 PUSH                            FEF|4          ; #'FOO
14 (AUX) COMPLEX-CALL-TO-PUSH
15 RETURN                          PDL-POP
```

The call-info argument has room for 63 values to be returned (bits 14–19) which is meaningful only when the return type is 1 (see Figure 22-1). The return type of 3 is used when the caller does not know how many values it wants (as in Example 17).

The **apply** function is also compiled using a complex call. Consider the following example:

Example 16:

```
(defun foo (a b &rest c)
 (apply #'format t a c)
 b)
```

The disassembled code appears as follows:

```
 8 SET-T                           PDL-PUSH
 9 PUSH                            ARG|0          ; A
10 PUSH                            LOCAL|0        ; C
11 PUSH-NUMBER                     67
12 PUSH                            FEF|3          ; #'FORMAT
13 (AUX) COMPLEX-CALL-TO-INDS
14 RETURN                          ARG|1          ; B
```

Note that bit 6 of the call-info word is set, indicating that this is an APPLY operation. Thus, the microcode passes the next three values on the stack (`t`, `a`, and `c` in this example) to the first argument to this function (`#'format`).

Also note that in instruction 10, the address `LOCAL|0` is used to access the &rest argument.

The **catch** special form is also handled specially by the compiler. The following is a simple example of **catch**:

Example 17:

```
(defun a ()
 (catch 'foo (bar)))
```

The disassembled code appears as follows:

```
14 PUSH                                 FEF|3           ; 'FOO
15 PUSH                                 FEF|4           ; '21
16 SET-NIL                              PDL-PUSH
17 (AUX) %OPEN-CATCH-MULTIPLE-VALUE
18 PUSH                                 FEF|5           ; '12288
19 PUSH                                 FEF|6           ; #'BAR
20 (AUX) COMPLEX-CALL-TO-PUSH
21 (AUX) %CLOSE-CATCH
22 (AUX) RETURN-N
```

The `(AUX) %OPEN-CATCH-MULTIPLE-VALUE` takes three arguments on the stack. Instruction 14 pushes the first argument, which is the **catch** tag 'FOO. Instruction 15 pushes the second argument, '21, which is the restart for the program counter (PC). This restart PC is the location in the function `a` that should be branched to if this **catch** is **thrown** to. Instruction 16 pushes the third argument, the symbol **nil**. This argument indicates that this `%OPEN-CATCH-MULTIPLE-VALUE` is expecting an unknown number of values. If the `%OPEN-CATCH-MULTIPLE-VALUE` knew the number of values being returned by the **throw**, then this argument would reflect that number. A value of 0 for the third argument indicates that all values returned are to be ignored.

The `(AUX) %OPEN-CATCH-MULTIPLE-VALUE` instruction receives its three arguments and saves both the state of the stack and the state of the special-variable binding so that they can be restored should a **throw** occur. Thus, instructions 14 through 17 start a **catch** block, and the rest of the function passes its two arguments. The **catch** form itself simply returns the value of its second argument; but if a **throw** happens during the evaluation of the `(bar)` form, then the stack is unwound and execution resumes at instruction 21.

Long Branches

22.4.6.3 Long branch instructions are similar to the short branches (except for **LONG-PUSHJ**). However, long branches use two macroinstruction words; the second word contains the new PC offset from the start of the FEF, rather than a signed relative displacement. Table 22-7 lists all the long branch instructions.

Table 22-7

Long Branch Instructions

LONG-BR	LONG-BR-NOT-ZEROP
LONG-BR-ATOM	LONG-BR-NULL
LONG-BR-NOT-ATOM	LONG-BR-NULL-ELSE-POP
LONG-BR-NOT-NULL	LONG-BR-SYMBOLP
LONG-BR-NOT-NULL-ELSE-POP	LONG-BR-ZEROP
LONG-BR-NOT-SYMBOLP	LONG-PUSHJ

LONG-PUSHJ is used to implement a macroinstruction subroutine call. The second word contains the new PC offset as described previously. The return PC (the relative address of the instruction after a **LONG-PUSHJ**) is saved on the stack. Control is then transferred to the macroinstruction stream at the new PC and returns via a **POPJ** instruction (a simple aux op). This instruction pops the return PC saved on the stack by **LONG-PUSHJ** and resumes execution where the **LONG-PUSHJ** left off.

AUX Op With Count Field

22.4.6.4 There are three instructions in this group: **POP-PDL**, which pops values off the stack, **UNBIND**, which unbinds special variables, and **RETURN**, which returns values from the stack as the result of the current function. Each of these instructions has a count field that specifies the number of values, up to a maximum of 63.

Module Operations

22.4.7 These instructions are similar to misc ops, except that they tend to be specific for certain applications or environments, and the presence of the microcode that implements a module is not required.

The operations to be performed and the module number are encoded in the instruction. Support is provided for 64 modules with eight operations for each module. The currently assigned modules and their instructions are listed in Table 22-8 in the form *(module)instruction*.

Table 22-8

Module Operation Instructions

(W:)%DRAW-CHARACTER	(W:)%DRAW-SHADED-RASTER-LINE
(W:)%DRAW-RECTANGLE	(W:)%DRAW-SHADED-TRIANGLE
(W:)%DRAW-STRING	

(MOUSE)%SET-MOUSE-SCREEN
(MOUSE)%OPEN-MOUSE-CURSOR

NOTE: The functions that are microcoded are subject to change.
Functions may be added to or deleted from this list without notice.

MAINTAINING LARGE SYSTEMS

23

Introduction

23.1 When a program grows large, it is often desirable to split it into several files and organize the parts so that things are easier to find. It is also useful to break the program into smaller pieces that are more convenient to edit and compile. It is particularly important to avoid the need to recompile all of a large program every time any portion of it changes. If the program is broken up into many files, usually only the files that have changes in them need to be recompiled.

The apparent drawback to splitting up a program is that more commands are needed to manipulate it. To load the program, you now must load several files separately, instead of simply loading one file. To compile the program, you must determine which files need compilation by verifying which have been edited after they were last compiled; then you must compile these files.

Even more complicated are the interdependencies between files. You might have a file called DEFS that contains macro definitions (or flavor or structure definitions), and functions in other files might use these macros. Thus, in order to compile any of these other files, you must first load the DEFS file into the Lisp environment so that the macros are defined and can be expanded at compile time. You must remember this step whenever you compile any of these files. Furthermore, if DEFS has changed, other files of the program may need to be recompiled because the macros may have changed and need to be reexpanded.

This section describes the *system* facility, which takes care of all these requirements for you. This facility allows you to define a set of files as a *system*, using the **defsystem** macro described below. This system definition indicates which files make up the system, which ones depend on the presence of others, and so on. You put this system definition into its own file, and then you merely load this file and the Lisp environment knows about your system and what files are in it. You can then use the **make-system** function (described later in this section) to recompile all the files that need compiling, load in all the files of the system, and so on.

The system facility is very general and extensible. This section explains how to use and how to extend the system facility. It also explains the *patch* facility, which lets you conveniently update a large program with incremental changes.

Defining a System

23.2 The following paragraphs describe how you define a system.

defsystem *name* {(*keyword* {*arg*}*)}* Macro

This macro defines a system called *name*, which is an unevaluated symbol. The options selected by the keywords are explained in detail later. In general, they fall into two categories: properties of the system and *transformations*. A transformation is an operation such as compiling or loading that takes one or more files and does something to them. You can think of a system's transformations as *code* that specifies how to build the system. The **make-system** function, in order to make a system, locates the system object representing this system and executes its transformation *code*. The simplest system is a set of files and a transformation to be performed on them.

Each option to **defsystem** is supplied as a list. The first item in each list is an option keyword followed by its arguments.

- (**:name** *name-string*) — The argument to this option is a string that specifies the full name for the system, for use in printing the herald.
- (**:short-name** *name-string*) — The argument to this option is a string that specifies an abbreviated name used in constructing disk label comments and in patch filenames for some file systems.
- (**:nicknames** {*name-string*}*) — Each *name-string* is made a synonym for the system name and can be used interchangeably.
- (**:output-version** {**:same**|**:newest**|**:higher**|**:ask-same**|**:ask-higher**}*) — This option describes how the version numbers of the output file should be created. The argument is passed to the compiler and behaves in accordance with the **compiler:*output-version-behavior*** variable. By default, this option has the value **:same**, which causes **compile-file** to create compiled output files, each with the same version number as the corresponding input file. When this argument is **:newest**, object files are written with a version number one greater than the highest version currently existing.

  Note that if you change the way you maintain your files, you may need to delete some object files for **make-system** to behave correctly. For instance, if you previously maintained your files using the **:newest** argument, your object file version numbers may become larger than the corresponding source file version numbers. If you then switch to **:same**, you may find that some source files are not being compiled using the **:compile** option. Even though a **:recompile** would compile the file, the generated object file may not be the greatest version number and thus not be loaded.
- (**:compile-defsystem**) — This option compiles and loads your most recent Lisp **defsystem** file. The default is to read the most recent definition (whether xld or lisp) and not to compile the **defsystem** file.
- (**:component-systems** {*name*}*) — The arguments to this option are the names of other systems used to make up this system. Performing an operation on a system with component systems is equivalent to performing the same operation on all the individual systems.
- (**:package** *name*) — The argument to this option is a package name that specifies the package in which transformations are performed. A package specified here overrides one in the file attribute (-*-) line at the beginning of the file in question.

  The preferred way to specify a package is with the file attribute line, which avoids the need for this option. Use of this option should be limited to special cases; for example, when debugging, you might want to put symbols in a different package, or the source files may not have file attribute lines. Note that this option can be locally overridden with the **:module** option.

- (**:pathname-default** *pathname-string*) — Gives a local default within the definition of the system for strings to be parsed into pathnames. Typically, this option specifies the directory when all the files of a system are on the same directory.

- (**:default-output-directory** *pathname-string*) — This option overrides **:pathname-default** for compiler and output loading. Explicit output file specification in the module definition overrides this default.

- (**:warnings-pathname-default** *pathname-string*) — Gives a default for the file to use for storing compiler warnings when **make-system** is used with the **:batch** option.

- (**:patchable** [*patch-directory-string* [*patch-node-name*]]) — Makes the system a patchable system (see paragraph 23.10, The Patch Facility). An optional argument specifies the directory in which to put patch files. The default is the **:pathname-default** of the system. The *patch-node-name* argument is the prefix for the filename of the patches. If the **:pathname-default** option is not used, the *patch-node-name* argument is ignored and the system name is used.

- (**:initial-status** {**:released**|**:experimental**|**:broken**|**:obsolete**}) — Specifies what the status of the system should be when **make-system** is used to create a new major version. The default is **:experimental**. See paragraph 23.12, System Status, for details.

- (**:not-in-disk-label**) — Prevents the **print-herald** function from displaying systems that use this option.

- (**:module** *name* {*module-spec*}* {*option*}*) — Assigns the symbol *name* to a set of files within the system. This *name* can then be used instead of repeating the filenames. The *module-spec* can be any of the following:

  - A string — This is a filename.

  - A symbol — This is a module name. It stands for all of the files that are in that module of this system.

  - An *external module component* — This is a list of the form (*system-name module-names...*) to specify modules in another system. It stands for all of the files that are in all of these modules.

  - A list of *module components* — A module component is any of the above, or it can be a list of filenames. This latter case occurs when the names of the input and output files of a transformation are not related according to the standard naming conventions, for example, when an object file has a different name or resides on a different directory than the source file. The filenames in the list are used from left to right; thus, the first name is the source file. Each filename after the first in the list is defaulted from the previous one in the list. To avoid syntactic ambiguity, this convention is allowed as a module component but not as a module specification. See the two examples of **defsystem** below for the different kinds of module definitions.

The **:module** clause has two keyword options:

- **:package** — This option provides a way to locally override the **defsystem :package** option, which affects only the files specified in the **:module** definition. Again, use of this option should be limited to special cases.

  The value of this keyword is a package name. The name can be a string, a symbol, or a quoted symbol. If a string is used, you probably should use all capital letters. For example:

  (:module *module-name module-spec* :package "USER")

- **:never-ship-p** — If the value of this keyword is true, then the files identified by this module are marked as special files that are *not* to be shipped with this system (if this system is ever shipped to another party). See the description of **copy-system** in paragraph 23.9, Copying a System, for more details. For example:

  (:module *module-name module-spec* :never-ship-p t)

The following are examples of all the possible *module-spec* definitions:

```
(:module module-name-1 "foo")          ; A single file. The host and directory are
                                       ; defaulted.

(:module module-name-2                 ; Three files.
  ("foo" "frob" "lm:bar;baz"))

(:module module-name-3                 ; Two files, each with a separate output
  (("foo" "lm:binary;")                ; directory. That is, source comes from the
   ("bar" "lm:binary;")))              ; default directory and the object file goes to
                                       ; "lm:binary;".

(:module module-name-4                 ; Includes module-name-1 of this system
  module-name-1)                       ; for module-name-4.

(:module module-name-5                 ; Includes modules 2 and 3 for
  (module-name-2 module-name-3))       ; module-name-5.

(:module module-name-6                          ; Includes these modules of
  (system-bar SB-module-1 SB-module-2))         ; system-bar as module-name-6.

(:module module-name-7                 ; They can all be intermixed in a single spec.
  ("foo"
  ("bar" "lm:binary;")
  some-module-name
  (some-module-name another-module-name)
  (system-bar module-1 module-2)))
```

The following are two examples of **defsystem**:

```
(defsystem mysys
 (:compile-load ("AI: BRUCE; PROG1"
                 "AI: BRUCE2; PROG2")))

(defsystem bar
 (:pathname-default "AI:EXPERT;")
 (:module reader-macros "RDMAC")
 (:module other-macros "MACROS")
 (:module main-program "MAIN")
 (:compile-load reader-macros)
 (:compile-load other-macros (:fasload reader-macros))
 (:compile-load main-program (:fasload reader-macros other-macros)))
```

The first example defines a new *system* called **mysys** that consists of two files, both of which are to be compiled and loaded. The second example has two levels of dependency. The `reader-macros` form must be loaded before `other-macros` can be compiled. Both `reader-macros` and `other-macros` must then be loaded before `main-program` can be compiled. This style of noting prerequisites is called *transformation dependencies*, which are discussed in paragraph 23.3, Transformations.

sys:set-system-source-file *system-name filename* Function

This function specifies which file contains the **defsystem** for the system indicated by *system-name*. The *filename* argument can be a pathname object or a name-string.

Sometimes it is useful to say where the definition of a system can be found without taking time to load that file. If **make-system** is ever used on this system, the file whose name has been specified is loaded automatically. You may want your saved system image to contain these definitions for all your application systems.

If the **defsystem** has not been loaded and this function has not been used to set up a pointer to the file, **make-system** reads the following file trying to find a **defsystem** definition SYS:SITE;*system-name*.SYSTEM, where *system-name* is the **defsystem** name.

Transformations

23.3 Transformations are of two types: simple and complex. A simple transformation is a single operation on a file, such as compiling or loading it. A complex transformation takes the output from one transformation and performs another transformation on it, such as loading the results of compilation.

The general format of a simple transformation is as follows:

(*name input* [*dependencies condition*])

The transformation *name* is to be performed on all the files in *input* or on all the output files of the other transformation. The *input* is usually a module specification or another transformation whose output is used.

Dependencies and conditions are explained more fully in paragraphs 23.3.1, Dependencies, and 23.3.2, Conditions.

Dependencies

23.3.1 A *dependency* declares that all of the indicated transformations must be performed on the indicated modules before the current transformation itself can take place. The *dependencies* argument is a *transformation specification*, which is either a list or a list of such lists. For example:

(*transformation-name* {*module-name*}*)

A *module-name* is either a symbol that is the name of a module in the current system or an external module component:

(*system-name* {*module-name*}*)

Thus, in the `bar defsystem` example in paragraph 23.2, Defining a System the `reader-macros` module must have the `:fasload` transformation performed on it before the `:compile-load` transformation can be performed on `other-macros`.

The dependency has to be a transformation that is explicitly specified as a transformation in the system definition, not merely an action that might be performed by anything. That is, if you have a dependency (`:fasload foo`), it means that (`:fasload foo`) is a transformation of your system, and you depend on that transformation; it does not simply mean that you depend on `foo` being loaded. Furthermore, (`:fasload foo`) cannot be an implicit piece of another transformation. For example, the following is correct and operable:

```
(defsystem foo
 (:module foo "FOO")
 (:module bar "BAR")
 (:compile-load (foo bar)))
```

However, the following does not work:

```
(defsystem foo
 (:module foo "FOO")
 (:module bar "BAR")
 (:module blort "BLORT")
 (:compile-load (foo bar))
 (:compile-load blort (:fasload foo)))
```

The preceding example does not work because of the way **defsystem** compares *dependencies* against previously seen *transformations*; that is, the module arguments `foo` and `(foo bar)` do not agree. You must write the following instead:

```
(defsystem foo
 (:module foo "FOO")
 (:module bar "BAR")
 (:module blort "BLORT")
 (:compile-load foo)
 (:compile-load bar)
 (:compile-load blort (:fasload foo)))
```

Conditions

23.3.2 The *condition* predicate specifies when the transformation should take place.

In general, any function that takes the same arguments as the transformation function (such as **compile-file**) and returns non-**nil** if the transformation needs to be performed can be used in this place as a symbol. Each transformation is defined with an appropriate default, so the need to specify a condition is somewhat limited. The most common need is satisfied by the **:compile-load-init** transformation. This transformation is explained in paragraph 23.6, Adding New Options for **defsystem**, and can be used as a pattern for defining new transformations or for simply finding out how the condition function can be used.

Most Used Transformations

23.3.3 For each of the following transformations, the *dependencies* default to **nil**. The defined simple transformations are as follows:

- (**:fasload** *module-spec* [*dependencies*] [*condition*]) — Calls the **fasload** function to load the indicated files, which must be object files whose pathnames have the **:xld** canonical type. (For more information about canonical types, see the *Explorer Input/Output Reference* manual.)

  The *condition* defaults to **sys:file-newer-than-installed-p**, which is true if the file has a newer version on disk than was read into the current environment.

- (**:readfile** *module-spec* [*dependencies*] [*condition*]) — Calls the **readfile** function to read in the indicated files, whose names must have the **:lisp** canonical type. Use this for files that are not to be compiled. The *condition* defaults to **sys:file-newer-than-installed-p**.

- (**:compile** *module-spec* [*dependencies*] [*condition*]) — Calls the **compile-file** function to compile the indicated files, whose names must have the **:lisp** canonical type. The *condition* defaults to **sys:file-newer-than-file-p**.

- (**:skip** . *transformation*) — The **:skip** transformation tells **make-system** to skip over each of the specified transformations entirely, unless some other transformation depends on the transformation in question. The **:skip** transformation can precede any legitimate transformation specification. For example:

  ```
  (:module A "foo")
  (:module B "bar")
  (:skip :compile-load A)
  (:compile-load B (:fasload A))
  ```

  This examples specifies that files in module A are to be skipped (that is, they will *not* be compiled or loaded) unless something in module B needs to be compiled—at which point this code will compile any files in A that need to be compiled and will load any file in A that needs to be loaded, all *before* compiling and loading files in module B.

- (**:auxiliary** *module-spec*) — Calls the **probe-file** function to probe the indicated files. Any file type can be used, but the default is **:lisp**. This transformation is useful for including in a system any files that have no other transformations performed on them (such as data files). This transformation merely verifies that the files exist, and if they do not, a warning is issued.

The following is a special simple transformation:

- (**:do-components** *dependencies*) — This transformation inside a system with component systems causes the *dependencies* to be performed before anything in the component systems. This option is useful when you have a module of macro files used by all of the component systems.

The defined complex transformations are as follows:

- (**:compile-load** *module-spec compile-dep load-dep compile-cond load-cond*) — This transformation is the same as the following:

(**:fasload** (**:compile** *module-spec compile-dep compile-cond*) *load-dep load-cond*)

This is the most commonly used transformation.

- (**:compile-load-init** *module-spec add-dep* [*compile-dep* [*load-dep*]]) — Files defined by *module-spec* are compiled and loaded if either of the following is true:
  - The source files in the module itself have changed. If these files are all that has changed, the behavior is like a **compile-load** of that module.
  - If the creation date of the source files defined by *add-dep* are newer than the object files defined by *module-spec*, then the *module-spec* source files are all recompiled.

The following are the arguments for **:compile-load-init**:

module-spec — An existing module.

add-dep — Any legitimate module specification. Usually, this argument is a list of other modules in this **defsystem**.

compile-dep — Compile dependencies to be performed before anything in the module is compiled. Typically, this argument should be a **:fasload** of the same modules in *add-dep* to ensure that those files are loaded before the module files are compiled.

load-dep — Load dependencies to be performed before anything in the module is loaded.

This transformation is ideal for ensuring that modules using macros defined in another module are recompiled when those macros are changed. It is also useful for recompiling flavor methods in which you have a module containing a single file that executes a **compile-flavor-methods** for all flavors that should be combined at load time. When there is a change to any of the modules that contain associated flavor and method definitions, the flavor methods are recombined. Consider the following example:

```
(defsystem my-application
   (:name "MA")
   (default-pathname "AI:MA;")
   (:module flavor-a ("foo" "bar"))
   (:module flavor-b "baz")
   (:module combine "compile-flavor-methods")

   (:compile-load    flavor-a
   (:compile-load    flavor-b)
   (:compile-load-init combine
                  (flavor-a flavor-b)           ; Additional dependencies
     (:fasload flavor-a flavor-b))              ; Compile dependencies
```

This form recompiles `"compile-flavor-methods"` if the source file for `"compile-flavor-methods"` changes or if the source files for `"foo"`, `"bar"`, or `"baz"` change. It also ensures that files for `flavor-a` and `flavor-b` are loaded before any compilation of the `combine` module.

More About Transformations

23.4 As explained in the previous examples, each filename in an input specification can be a list of strings when the source file of a program differs from the object file in more than just the file type. In this sense, it is as if the last filename specified (in most cases only one name is specified) is repeated as many times as necessary at the end of the list, and the type specification is adjusted depending on the type of transformation. Each simple transformation takes a number of input filename arguments and a number of output filename arguments. As the transformation is performed, arguments are taken from the front of the filename list. The output arguments are remembered as input arguments to the next higher transformation if this is a complex transformation.

Consider the following example:

```
(:module prog (("AI:BRUCE;PROG" "AI:BRUCE2;PROG")))
(:compile-load prog)
```

In this case, since `:compile-load` is equivalent to `(:fasload (:compile` *module-spec* `...))`, `prog` is given as the input to the **:compile** transformation, and the output from this transformation is given as the input to the **:fasload** transformation. The **:compile** transformation takes one input filename argument (the name of a Lisp source file) and one output filename argument (the name of the object file). The **:fasload** transformation takes one input filename argument (the name of an object file) and no output filename arguments. Thus, for the first and only file in the `prog` module, the `:module` argument is equivalent to the following:

```
(:module prog (("AI:BRUCE;PROG" "AI:BRUCE2;PROG" "AI:BRUCE2;PROG")))
```

In this form, the first filename (`"AI:BRUCE;PROG"`) is the compiler input argument, the second filename is the compiler output argument, and the third filename is the loader input argument. Of course, **compile-file** defaults the output file type to be xld because this is an object file. The description of the function **sys:define-simple-transformation** in paragraph 23.6, Adding New Options for **defsystem**, provides additional insight on how arguments are processed.

Note that dependencies are not *transitive* or *inherited*. For example, if module `x` depends on macros defined in module `y` and therefore needs `y` to be loaded to compile, and `y` has a similar dependency on `z`, then `z` need not be loaded for the compilation of `x`. Transformations with these dependencies are written as follows:

```
(:compile-load x (:fasload y))
(:compile-load y (:fasload z))
```

To say that compilation of `x` depends on both `y` and `z`, you instead write the following:

```
(:compile-load x (:fasload y z))
(:compile-load y (:fasload z))
```

If, in addition, `x` depended on `z` (but not `y`) during loading (perhaps `x` contains **defvars** whose initial values depend on functions or special variables defined in `z`), you write the transformations as follows:

```
(:compile-load x (:fasload y z) (:fasload z))
(:compile-load y (:fasload z))
```

Summary of Compiler Conditions and Dependencies

23.5 The following paragraphs summarize the **defsystem** requirement that the compiler must *see x before y* whether *x* and *y* are Lisp forms in a file, or whether they are in files presented to the compiler by the **make-system** function based on a **defsystem** declaration. The compiler will warn you about many of these types of conflicts if conflict is seen during a single compilation. However, a conflict caused across separate compilations cannot be seen.

A possible problem arises from back-to-back compilation. The second compilation always sees *x* before *y*, even though some of the *x*'s are old definitions and some are new. Because of this restriction, the following situations are true only for the first compilation and load following a cold boot.

- Generally, you should place definitions so that the compiler sees them before it sees any references to those definitions.

- For efficiency, place **deftype** before you use the type it defines. However, execution is correct regardless of the order.

- The **defresource**, **defsignal**, and **defsignal-explicit** macros have *no* ordering constraints for either efficiency or correct execution.

- For correct execution of the special forms that you define by using &quote in a function's lambda list:

    - Always place the special form definition so that the compiler sees it *before* it sees any calls to that special form. Otherwise, the function is called at run time with its quoted argument(s) evaluated.

    - Recompile all calls to that special form if you change which arguments to the function are quoted. Otherwise, the function is called at run time with the wrong arguments evaluated. You can find a call to a macro by using **who-calls**.

- If you define functions that use the &functional lambda-list keyword, then these functions should be seen before any calls to them. Although no error would be created if they were not seen in this order, the compiler would not know that it is acceptable to precompile any corresponding arguments.

- For correct execution of macros you define by using **defmacro**:

    - Always place the **defmacro** form so that the compiler sees it *before* it sees any calls to that macro. Otherwise, the code attempts to illegally **funcall** a macro at run time.

    - Recompile all calls to that macro if you change the macro's definition so that it expands differently. Otherwise, unrecompiled code still uses the old macro definition.

    - Files that use macros should be compiled with a **compile-load-init** transformation.

- Of the functions that you have specified by **proclaim** to be **inline**:
  - For efficiency's sake only, place the **proclaim**, the **defun**, and the calls to that function so that the compiler sees them in that order. Otherwise, the calls are treated as ordinary function calls rather than as being expanded inline.
  - For correct execution, recompile all places where the function was expanded inline if you change the function so that it expands differently. Otherwise, unrecompiled code still uses the old function definition. You can find a call to an **inline** function by using **who-calls**. Look for those marked `Used as macro` (which really means `Used inline`).

NOTE: If, and only if, you have provided an **optimize** declaration with **safety** equal to 0 and **speed** greater than **size** and you have not provided a **proclaim notinline**, then the compiler can choose—at its discretion—to make short functions inline even though you did not **proclaim** them **inline**. Use **who-calls** to check whether this has happened to a particular function. If the function is listed as `Used as a macro`, then this function was expanded inline (whether by a **defmacro**, **defsubst**, or **proclaim inline**).

- Of substitutable functions you have defined with **defsubst**:
  - For efficiency's sake, place the **defsubst** so that the compiler sees it *before* the compiler sees any simple calls to that subst. Otherwise, the calls are treated as ordinary function calls rather than as being expanded inline.
  - For correct execution, place the **defsubst** so that the compiler sees it *before* its sees any uses of the subst as a *generalized variable* (that is, as the place argument of a **setf**, **incf**, **decf**, and so on). Otherwise, a warning appears indicating that the compiler does not know how to **setf** the form.
- For functions defined automatically for structures you have defined using **defstruct**:
  - If you are in Common Lisp mode and have accepted the defaults for callable constructors, for callable accessors, and for no alterant, then treat the **defstruct** as a **defsubst**.
  - For correct execution in *all* other cases (including Zetalisp mode), treat the **defstruct** as a **defmacro**.

NOTE: Recall that a **defstruct** actually defines a number of functions. If *struct* is the name of a structure and *slot* is the name of a slot in that structure, then **defstruct** creates all of the following:

- A **funcallable** and **setf**able accessor function *struct-slot*
- A **funcallable** constructor function **make-***struct*
- A **funcallable** predicate *struct*-**p**
- A **funcallable** copier function **copy-***struct*

- For correct execution of variables you have proclaimed special or defined with **defvar** or **defparameter**:
  - Always place the **proclaim**, **deconstant**, **defvar**, or **defparameter** so that the compiler sees it *before* seeing any uses of that variable. Otherwise, a compiler warning appears indicating that the compiler does not recognize the variable and assumes it to be special.
  - If you modify your code so that a variable previously proclaimed/ defined to be special is no longer special, you must **proclaim** that variable **unspecial** and then recompile all portions of the code using that variable. To find out what needs to be recompiled, use the **who-uses** function.
- Of constants defined with **defconstant**:
  - For efficient operation, place the **defconstant** so that the compiler sees it before seeing any uses of that constant. Otherwise, the value is accessed as a standard **defparameter** value at run time.
  - For correct execution, if a **defconstant** value is changed, always recompile all places where a constant is expanded inline. Otherwise, the code that has not been recompiled uses the previous value. The **who-uses** function will not find such uses of **defconstant** that need to be recompiled. You must use **compile-load-init** instead.
- Keep the following points in mind for functions defined with **defun-method**, defined by placing a function definition inside a **declare-flavor-instance-variable** macro, or defined by placing a `declare :self-flavor` in a function definition (collectively referred to as *pseudo-methods* in the next paragraphs):
  - For efficient operation, place the pseudo-method so that the compiler sees it before the compiler sees any calls to it. Otherwise, the function suffers extra runtime entry overhead each time it is called.
  - For correct execution, you must call a pseudo-method only from within a method or another pseudo-method of the parent flavor. Otherwise, the pre-initialized environment the pseudo-method is expecting upon entry at run-time does not exist and you get arbitrarily strange results.

  - For correct execution, you must place the parent **defflavor** of the pseudo-method so that the compiler sees it before it sees the pseudo-method. Otherwise, the compiler mistakes references to the flavor's instance variables in the pseudo-method as references to free variables.

- For correct execution of functions you have defined using **defmethod** or **defwrapper** place the parent **defflavor** of the **defmethod** or **defwrapper** so that the compiler sees the **defflavor** before it sees the **defmethod** or **defwrapper**. Otherwise, the compiler has no place to record the **defmethod** or **defwrapper** and mistakes references to the flavor's instance variables in the **defmethod** or **defwrapper** as references to free variables.

- For efficient execution, place a **compile-flavor-methods** after all **defmethods** of the parent flavor. Otherwise, the *first* **make-instance** of that flavor suffers a one-time delay.

Adding New Options for defsystem

23.6 Before the forms associated with adding new options for **defsystem** are described, you should know that these features may be altered/enhanced in the future; therefore, any options you add for **defsystem** may require changes to retain compatibility with future releases.

Options to **defsystem** are defined as macros on the **sys:defsystem-macro** property of the option keyword. Such a macro can expand into an existing option or transformation, or it can have side effects and return **nil**. These macros can use several variables; however, **sys:*system-being-defined*** is the only one of general interest.

sys:*system-being-defined* Variable

This variable points to the internal data structure that represents the system currently being constructed. It contains an instance of the system **defstruct** that is bound when **defsystem** is processed. To view this variable, use the Inspector facility (see the *Explorer Tools and Utilities* manual).

sys:define-defsystem-special-variable *variable value* Macro

This macro causes *value* to be evaluated and *variable* to be bound to the result during the expansion of the **defsystem** macro. This macro allows you to define new variables similar to the one described above. For example:

```
(sys:define-defsystem-special-variable *start-time*
  (get-universal-time))
```

If you write an extension to the **defsystem** macro, it has access to the special variable ***start-time***, which is initialized to the Universal time when the **defsystem** is evaluated.

sys:define-simple-transformation *name transform-function default-condition-function input-file-types output-file-types* &optional *pretty-names compile-like load-like transformation-input-type* Macro

This macro provides the most convenient way to define a new simple transformation. The *name* argument should be a keyword, *transform-function* performs the implied operation, and *default-condition-function* accepts the same pattern of arguments that the *transform-function* does but returns a non-**nil** value if the transformation should be performed.

For example:

```
(sys:define-simple-transformation :compile
      sys:qc-file-1
      sys:file-newer-than-file-p
      ("LISP") ("XLD"))
```

The *input-file-types* and *output-file-types* are how a transformation specifies the number of input filenames and output filenames it should receive as arguments, in this case one of each. All input arguments are assumed to precede all output arguments. They also, obviously, specify the default file type for these pathnames. The **sys:qc-file-1** function is mostly like **compile-file**, except for its interface to packages. It takes input-file and output-file arguments.

The *pretty-names* argument specifies how messages printed for the user should print the name of the transformation. It can be a list of the imperative (Compile), the present participle (Compiling), and the past participle (compiled). Note that the past participle is not capitalized because, when used, it does not come at the beginning of a sentence. The *pretty-names* can be simply a string, which is taken to be the imperative, and the system conjugates the participles itself. If *pretty-names* is omitted or **nil**, it defaults to the name of the transformation.

The *compile-like* and *load-like* arguments indicate when the transformation should be performed. Compile-like transformations are performed when either the **:compile** or the **:recompile** keyword is given to **make-system**. Load-like transformations are performed unless the **:noload** keyword is given to **make-system**. By default, *compile-like* is non-**nil**, but *load-like* is **nil**.

The *transformation-input-type* argument is used to indicate what transformation (if any) can be used as input to the transformation being defined. For example, the **:fasload** transformation can have **:compile** as its input. This argument is necessary for any transformation that can accept another transformation as its input.

Complex transformations are defined as macro expansions. The following example shows how the complex transformation **:compile-load** might be defined:

```
(defmacro (:property :compile-load sys:defsystem-macro)
   (input &optional com-dep load-dep com-cond load-cond)
  `(:fasload (:compile ,input ,com-dep ,com-cond) ,load-dep ,load-cond))
```

Note that the load dependencies are defined before the compile conditions for the macro arguments, but the macro expansion converts the `:compile-load` transformation to two simple transformations, one nested inside the other.

The **compile-load-init** transformation could be defined as follows:

```
(defmacro (:property :compile-load-init sys:defsystem-macro)
    (input add-dep &optional com-dep load-dep &aux function)
 (setf function (let-closed ((*additional-dependent-modules* add-dep))
          'compile-load-init-condition)))

(defun compile-load-init-condition (source-file object-file)
 (declare (special *additional-dependent-modules*))
 (or (sys:file-newer-than-file-p source-file object-file)
     (sys:other-files-newer-than-file-p
      *additional-dependent-modules* object-file))))
```

When this macro is used, the condition function that is generated returns non-**nil** in either of two situations:

- If **sys:file-newer-than-file-p** returns non-**nil** with the same arguments
- If any of the other files in **add-dep** (presumably a *module specification*) are newer than the object file

Thus, the file or module to which the **:compile-load-init** transformation applies is compiled if it, or any of the source files it depends on, has been changed and is then loaded under the normal conditions. In most (but not all) cases, **com-dep** is a **:fasload** transformation of the same files as specified by **add-dep**, so that all the files on which this one depends are loaded before it is compiled.

Making a System

23.7 To understand the **make-system** function description in this paragraph, refer to the following example:

```
(defsystem mysys
 (:compile-load ("AI: BRUCE; PROG1"
                 "AI: BRUCE2; PROG2")))
```

make-system *system-name* &rest *keywords* Function

This function does the actual work of compiling and/or loading. First, **make-system** checks whether the file that contains the defined system has changed since it was loaded. If so, **make-system** asks the user if it should load the latest version before continuing, so that it can use the latest system definition. (This happens only if the file is type lisp.) After loading this file, if necessary, **make-system** goes on to process the files that compose the system.

If **make-system** cannot find *system-name* by virtue of the **defsystem** being known or by means of **sys:set-system-source-file**, then **make-system** attempts to load the following file in the hope that it contains a system definition:

SYS:SITE;*system-name*.SYSTEM

The **make-system** function lists which transformations it is going to perform on which files, asks the user for confirmation, and then performs the transformations. Before each transformation, a message is printed, listing the transformation being performed, the file being processed, and the package. This behavior can be altered by keywords. If the system is being loaded (as opposed to merely compiling files), **make-system** attempts to load patches.

The following are the keywords recognized by **make-system**:

:noconfirm — Assumes a yes answer for all questions that would otherwise be asked of the user.

:nowarn — Causes redefinition warnings to merely be printed (no user response is required). Also turns on **:noconfirm**.

:selective — Asks the user whether to perform each transformation that appears to be needed for each file.

:silent — Avoids printing out each transformation as it is performed. Note that this option does not turn off redefinition warnings; it only turns off **make-system** queries.

:reload — Forces the reloading of all object files. For example:

```
(make-system 'foo :compile :reload)
```

This form compiles all the files that need compilation, but it also forces all files to be reloaded—even those that do not appear to need reloading.

:noload — Does not load any files except those required by dependencies. For use in conjunction with the **:compile** option.

:do-not-do-components — Prevents **make-system** from attempting to perform any transformations for dependent systems. The default is to perform transformations for dependent systems.

:compile — Compiles files if needed according to the transform condition function. The default is to load but not compile. Options can also be passed to the compiler here. The compiler options that are meaningful to pass are **:package**, **:verbose**, **:declare**, and **:suppress-debug-info**. Using **:output-file** is probably a mistake since **defsystem** normally makes implicit assumptions about where the output should be. For example, the following form causes the `:verbose` and `:package` keywords and values to be passed along to the compiler:

```
(make-system 'my-sys '(:compile :verbose t :package SYS))
```

In the default case, **:compile** causes the major version number to be incremented.

:recompile — Specifies compilation of all files, even those whose sources have not changed since they were last compiled. In these cases, the compile transform condition is implicitly true. Options can also be passed to the compiler here. The compiler options that are meaningful to pass are **:package**, **:verbose**, **:declare**, and **:suppress-debug-info**. Using **:output-file** is probably a mistake since **defsystem** usually makes implicit assumptions about where the output should be. For example:

```
(make-system 'my-sys '(:recompile :verbose t :package SYS))
```

In the default case, **:recompile** causes the major version number to be incremented.

:record — Specifies the recording of all version numbers of the files used to make this system. The system being made must be a patchable system. The information is kept in the file *patch-directory*;VERSION.LOG. It records the system name, the major version number, and the truenames of the files loaded. Afterwards, you can use the **:version** option to **make-system** to remake this particular major version of the indicated system.

(:version *major-version-number*) — Causes **make-system** to attempt to remake the system as it existed for *major-version-number*. You must have previously executed a **make-system** with the **:record** option to produce the *major-version-number*. Specifically, **make-system** loads the **defsystem** that was in effect for that major version, and it loads files associated with that **defsystem**. Then **make-system** reinitializes the patch system version so that patches are loaded. If *major-version-number* is not supplied, then the user is prompted for the major version to be made.

:no-load-patches — Prevents **make-system** from attempting to load patches. The default is to attempt to load patches.

:no-increment-patch — When given along with the **:compile** or **:recompile** option, disables the automatic incrementing of the major system version that would otherwise take place for a patchable system. See paragraph 23.10, The Patch Facility. Note that this option does not disable patch loading.

:increment-patch — Increments a patchable system's major version without performing any compilations. See paragraph 23.10, The Patch Facility.

:no-reload-system-declaration — Turns off the check for whether the file containing the **defsystem** has been changed. This file is loaded only if it has never been loaded before.

:batch — Allows a large number of compilations to be performed unattended. It acts like **:noconfirm** for questions, turns off *more-processing* and *fdefine-warnings* (see **inhibit-fdefine-warnings** in Section 12, Type Specifiers), and saves the compiler warnings in a file (prompting you for the name).

:defaulted-batch — Resembles **:batch** except that it uses the default for the pathname (specified with **defsystem**) in which to store warnings and does not ask the user to type a pathname.

:print-only — Prints only which transformations would be performed; does not actually perform any compiling or loading.

:noop — Is ignored. This keyword is useful mainly for programs that call **make-system** so that such programs can include forms such as the following:

```
(make-system 'mysys (if compile-p :compile :noop))
```

Adding New Keywords to make-system

23.8 The **make-system** keywords are defined as functions on the **sys:make-system-keyword** property of the keyword. The functions are called with no arguments. For example:

```
(defun (:property :beep sys:make-system-keyword) ()
  (beep))

(make-system 'mysys :beep)
```

Some of the relevant variables they can use are described in the following paragraphs:

sys:*system-being-made* Variable

This variable is the instance of the system structure that represents the system being made. To view this variable, use the Inspector facility (see the *Explorer Tools and Utilities* manual).

sys:*make-system-forms-to-be-evaled-before* Variable

This variable is a list of forms that are evaluated before the transformations are performed. You can view its contents with the Inspector facility (see the *Explorer Tools and Utilities* manual).

sys:*make-system-forms-to-be-evaled-after* Variable

This variable represents a list of forms that are evaluated after the transformations have been performed. Transformations can also push entries to this list.

sys:*make-system-forms-to-be-evaled-finally* Variable

This variable represents a list of forms that are evaluated by an **unwind-protect** when the body of **make-system** is exited, whether it is completed or not. Closing the batch warnings file is performed in this group of forms. Unlike the **sys:*make-system-forms-to-be-evaled-after*** forms, these forms are evaluated outside of the compiler warnings context.

sys:*query-type* Variable

This variable controls how questions are asked. Its normal value is **:normal**. A value of **:noconfirm** means no questions are asked, and a value of **:selective** asks a question for each individual file transformation.

sys:*silent-p* Variable

If this variable is true, no messages are printed out.

sys:*batch-mode-p* Variable

If this variable is true, the **:batch** or defaulted batch was specified on **make-system**.

sys:*redo-all* Variable

If this variable is true, all transformations are performed, regardless of the condition functions.

sys:*redo-load-type Variable

When this variable is bound to true, it forces all load transformations (those in the list **sys:*load-type-transformations***) to be performed.

sys:*file-transformation-list* Variable

This variable is a list of all the transformations that can be performed, including dependencies.

sys:*top-level-transformations* Variable

This variable is the list of transformation types that are to be processed during the current execution of **make-system**. This variable defaults to the *load-like* transformation types. Depending on various **make-system** options specified, the *compile-like* and other transformation types are added to this list before processing the file transformations for this system.

sys:*transformation-type-alist* Variable

This variable is an association list of all the transformation types known to **make-system**.

sys:*file-transformation-function* Variable

This variable represents the actual function that is called with the list of transformations that need to be performed. The default is **sys:do-file-transformations**.

sys:define-make-system-special-variable *variable value* &optional *defvar-p* Macro

This macro causes *variable* to be bound to *value* during the body of the call to **make-system**. This macro allows you to define new variables similar to those listed above. The *value* argument is evaluated on entry to **make-system**. If *defvar-p* is specified as (or defaulted to) **t**, *variable* is defined with **defvar** to make it special, but it is not given a global value. If *defvar-p* is specified as **nil**, *variable* belongs to another program (which presumably makes it special) and is not defined here.

The following simple example adds a new keyword to **make-system** called **:just-warn**, which means that **fdefine** warnings (see Section 16, Functions) regarding functions being overwritten should be printed out, but the user should not be queried:

```
(sys:define-make-system-special-variable
  inhibit-fdefine-warnings inhibit-fdefine-warnings nil)

(defun (:property :just-warn sys:make-system-keyword) ()
 (setf inhibit-fdefine-warnings :just-warn))
```

The **make-system** keywords can act directly when called, or they can produce their effect by pushing a form to be evaluated onto **sys:*make-system-forms-to-be-evaled-after*** or one of the other two similar lists.

In general, the only useful action is to set a special variable defined by **sys:define-make-system-special-variable**. In addition to the ones mentioned above, user-defined transformations may have their behavior controlled by new special variables, which can be set by new keywords. If you want to access the list of transformations to be performed, for example, the correct way is to set **sys:*file-transformation-function*** to a new function, which can then call **sys:do-file-transformations** with a possibly modified list. This is how the **:print-only** keyword works.

Copying a System

23.9 The following function is for copying the files of a system.

copy-system *system-name* &key **:from-host :to-host :intermediate-too :include-subsystems :ignore-never-ship :include-patch-files :system-version :output-version :overwrite :file-type** Function

This function is used to copy the files associated with the *system-name* from one machine to another. This can include patch files.

:from-host — This option identifies the originating host for the files. You must supply a value representing the host of origin, or accept the default host in the **defsystem** pathnames.

:to-host — This option identifies the destination host for the files. You must supply a value representing that host, or accept the default value of **lm**, meaning local machine.

:intermediate-too — When true (the default), this option copies the intermediate files generated by complex transformations. In almost all cases, the complex transformation is **:compile-load**, and files in question are the object files. If you do not want object files copied, enter **nil** for this keyword.

:include-subsystems — When true (the default), this option copies the files of component systems.

:ignore-never-ship — When true, this option copies all files marked with **:never-ship-p** in their **defsystem**. The default is **nil**.

:include-patch-files — When true (the default), this option copies the patch files.

:system-version — The value of this option should be a number that corresponds to a major version number of a previously recorded system version via the form (**make-system** *system* **:record**). The default value (**nil**) copies the current major version number.

:output-version — This option specifies the version pathname component for output files. This argument is passed to the file system copy function. The default is **:wild** to preserve source version numbers.

:overwrite — When true, this option overwrites existing output files if **:output-version** is **:wild**. The default is **nil**.

:file-type — This option is a list of file types to copy. For example, `'(:LISP)` copies only files that have the canonical LISP extension. The default value is **:all**, which means all types.

The Patch Facility

23.10 The patch facility allows the person who maintains the system to manage new releases of a large system and issue patches to correct bugs. It is designed to maintain both the Explorer system itself and applications systems that are large enough to be loaded and saved on a disk partition.

When a system of programs is very large, it needs to be maintained. Often problems are found and must be fixed, or other small changes need to be made. However, loading all the files that comprise such a system is time consuming. Thus, each user does not load all the files each time he or she wants to use the system. Rather, the files are loaded only once into a band, which is then saved away on a disk partition using **disk-save**. Users then boot this disk partition, copies of which may be distributed to many machines. However, because users do not remake the system every time they want to use it, they do not have all the latest changes.

The patch system is designed to solve this problem. A *patch* file is a file that, when loaded, updates the old version of the system to be functionally compatible with the later source files. Most often, patch files simply contain new function definitions; old functions are redefined to perform slightly altered tasks. Patch files are relatively small, so loading them does not take much time. You can even boot the saved environment, load up the latest patches, and then save the environment away to spare future users the trouble of even loading the patches. (Of course, more new patches can be made later; then, these must be loaded if you want to use the very latest version.)

Systems are patchable if their **defsystem** uses the **:patchable** option. For every patchable system, a series of patches can be made to that system. To access the latest version of the system, you should load the latest patches using **load-patches**. Sooner or later, the maintainer of a system will want to stop adding more patches and recompile everything, starting fresh.

A complete recompilation is also necessary when a system is changed in a farreaching way, which cannot be done with a small patch. For example, if you completely reorganize a program or change a number of names or conventions, you might need to completely recompile the program to make it work again. After the program has been completely recompiled, the old patch files are no longer suitable to use; loading them can even cause serious problems.

The state of a patchable system is tracked by labeling each version of a system with a two-part number. The two parts are called the *major version number* and the *minor version number*. The minor version number is increased every time a new patch is made. The major version number is increased when the program is completely recompiled, and at that time the minor version number is reset to zero. A patch is identified by the major and minor version number together. A complete system version is identified by the major version number, followed by a dot, followed by the minor version number. Thus, patch 1.2 is for major version 1 and minor version 2; it is followed by patch 1.3. When the entire system is recompiled, version 2.0 is created from scratch, and the previous patches are irrelevant because they fix old software.

To use the patch facility, you must define your system with **defsystem** (described in paragraph 23.2, Defining a System) and declare it as patchable with the **:patchable** option. When you load your system with **make-system** (described in paragraph 23.7, Making a System), you add it to the list of all systems present in the current band. The patch facility keeps track of which version of each patchable system is present and where the data about that system resides in the file system. This information can be used to update the Lisp environment automatically to the latest versions of all the systems it contains. Once a system is present, you can ask for the latest patches to be loaded, ask which patches are already loaded, and add new patches.

When the Explorer system is booted, it prints out a line of information telling you which patchable systems are present and which version of each system is loaded. It is followed by a text string containing any additional information requested by whoever created the current disk partition (see the **disk-save** function in paragraph 23.11, Saving to Disk).

Patch Version Information

23.10.1 The following functions return various bits of information about patch versions.

print-system-modifications {*system-name*}* Function

With no arguments, this function lists all the systems present in the booted environment and, for each system, all the patches that have been loaded. For each patch, it shows the major version number (which is always the same since a partition can contain only one major version for any given system), the minor version number, and an explanation of what the patch does, as typed in by the person who made the patch.

If **print-system-modifications** is called with arguments, only the modifications to the specified systems are listed.

sys:get-system-version &optional *system* Function

This function returns three values: the major and minor version numbers of the version of *system* currently loaded into the machine and the status. If this system is not present, **nil** is returned. The *system* argument defaults to System, meaning the base operating system.

sys:system-version-info &optional *brief-p* Function

If *brief-p* is **nil** (the default), this function returns a string giving information about which systems and which versions of the systems are loaded into the machine, and which microcode version is running. The string is derived from the values supplied at the time the systems were defined (**defsystem**). The following is a typical string produced by this function:

```
"System 1.3, EXPLORER STREAMER TAPE 5.0, microcode 006"
```

If *brief-p* is true, the function returns only the disk label herald description.

Patch Files and Patch Directories

23.10.2 The patch system maintains several different types of files in a directory associated with your system. The patch files are maintained automatically, but to help you know what they are and when they are obsolete (because they are associated with an obsolete version of your system), they are described in the following paragraphs.

The directory in which the patch files reside is specified to **defsystem** via the **:patchable** option. By supplying a directory name as an argument to the **:patchable** option of **defsystem**, you can name your own directory in which the patch files will reside. If you do not specify a value for **:patchable**, the directory takes its name from the value of **:pathname-default**.

Local Patch Directory and Files

23.10.2.1 If you are using the local patch directory, the file identifying the system's current major version has the following form:

host:dir;PATCH.PATCH-DIRECTORY

In this form, *host:dir* is provided by **:pathname-default**. This file is very small and is present only to record the current major version number.

For each major version of the system, there is a *patch directory file*, that takes the following form:

host:dir;PATCH-*maj*.PATCH-DIRECTORY

As before, *host:dir* is provided by **:pathname-default**, but in this form, *maj* is a number representing the major version number of the system. This file contains patch descriptions and is actually used as a locking mechanism to allocate the next patch number and to track which patches are released.

Then, for each minor version of the system, the source of the patch file itself has a name of the following form:

host:dir;PATCH-*maj-min*.LISP

Again, *host:dir* is provided by **:pathname-default**, but in this form, *maj* represents the major version number of the patch and *min* represents the minor version number of the patch.

User-Named Patch Directory and Files

23.10.2.2 If you have designated (with the **defsystem** function) a particular directory to hold the patch files, the file identifying the system's current major version has the following form:

host:dir;name.PATCH-DIRECTORY

In this form, *host:dir* has the value you supplied as an argument to **:patchable**. The *name* component is the name of the system; it is the same as the *name* supplied to **defsystem** when the system was defined.

The patch directory file for a user designated patch directory has the following form:

host:dir;name-maj.PATCH-DIRECTORY

Again, *host:dir* has the value you supplied as an argument to **:patchable**, and *name* is the name given to the system when it was defined. In this form, *maj* is a number representing the major version number of the system.

For each minor version of the system, the source of the patch file has a name of the following form:

host:dir; *name-maj-min*.LISP

Again, *host:dir* has the value you supplied as an argument to **:patchable**, *name* is the name given to the system when it was defined, and *maj* is the major version number for which this patch was created. In this form, *min* represents the minor version number of the patch.

Loading Patches

23.10.3 The following function is used to load patches.

load-patches {*option*}* Function

This function is used to bring the current environment up to the latest minor version of the currently loaded major version, for all systems present, or for certain specified systems. If any patches are available, **load-patches** offers to read them in. If no specific systems are named in *options*, **load-patches** updates all the patchable systems present in the environment.

The **load-patches** function returns **t** if any patches are loaded or **nil** otherwise.

The *option* argument is a list of keywords. Some keywords are followed by an argument. The following options are accepted:

a-system-name — This option loads the patches for this system.

:systems *list* — The value of *list* is a list of names of systems to load patches for. If this option is not specified, all systems are processed.

:selective — For each patch, **:selective** indicates what it is and then asks the user whether to load it. This is the default. If the user responds with P for PROCEED, the selective mode is turned off and the default response is implied for all subsequent situations where a question would otherwise be asked. That is, released patches are loaded and unreleased patches are not loaded.

:noconfirm, **:noselective** — These two options turn off **:selective**; all patches are loaded without asking for confirmation.

:verbose — This keyword prints an explanation of what is being done and can only be turned off by **:silent**. This is the default.

:silent — This turns off both **:selective** and **:verbose**. In **:silent** mode all necessary patches are loaded without printing anything and without querying the user for confirmation.

:force-unfinished — This loads patches that have not been finished yet, if they have been compiled. This option is useful for testing a patch before releasing it to all the users.

Currently **load-patches** is not called automatically, but the system may be changed to offer to load patches when the user logs in, in order to keep software updated.

Making Patches

23.10.4 Two editor commands are used to create patch files. During a typical maintenance session on a system, you make several changes to its source files. The patch system can be used to copy these changes into a patch file so they can be formally incorporated into the system to create a new minor version. A patch file can modify function definitions, add new functions, modify **defparameters** and **defvars**, or contain arbitrary forms to be evaluated, even including **loads** of new files.

The Add Patch Command

23.10.4.1 To make a patch, first modify the source file to reflect the desired change. If your **defsystem** uses a logical host name, then you should use the same logical host name when you use the Zmacs Find File command to ensure that the patch system uses the logical name also. Next use the Zmacs Add Patch command (see the *Explorer Zmacs Editor Reference* manual). You need to use this command for each top-level form that has changed. The first time you enter this command, you are asked which system you are patching, a new minor version number is allocated, and a patch buffer for this version is constructed. Repeatedly execute the Add Patch command until you have completed all necessary changes. You can include patches for several different source files in one patch session.

The patch file is constructed in an editor buffer. If you mistakenly perform the Add Patch command to something that does not work, you can select the buffer containing the patch file and delete it. Then you can perform the Add Patch command on the corrected version.

While you are making your patch file, the minor version number that has been allocated for you is reserved so that nobody else can use it. Thus, if two people are patching a system at the same time on different machines, they do not both get the same minor version number. Note that Zmacs does not allow you to concurrently make two distinct patches on the same machine.

The Finish Patch Command

23.10.4.2 After making and testing all of your patches, use the Zmacs Finish Patch command to install the patch file so that other users can load it. This command compiles the patch file if you have not done so yourself (patches are always compiled). It asks you for a comment describing the reason for the patch. The **load-patches** and **print-system-modifications** functions print these comments.

After finishing your patch, if you perform another Add Patch command, the patch facility again asks which system you are patching and starts a new minor version.

The Start Patch Command

23.10.4.3 You can also start a patch without adding anything to it. Use the Zmacs Start Patch command. This command does everything the Add Patch command does except that it does not add a patch region to the buffer; it only performs the initial bookkeeping. After starting a patch in this way, you can use the Add Patch command to add the changed definitions to the patch buffer.

The Resume Patch Command

23.10.4.4 If you wish to defer finishing the patch until a later session, save the editor buffer that contains the patch file and the source file(s) you have been modifying. In the next session, use the Zmacs Resume Patch command to reselect the patch. You must specify the minor version number of the patch you wish to resume (it would be wrong to assume that your patch is the most recent one, since someone else might have started one). Once you have done this, you are again in a position to perform the Add Patch, Finish Patch, or Cancel Patch command on this patch.

The Cancel Patch Command

23.10.4.5 You can cancel a finished patch by performing the Resume Patch command and then the Cancel Patch command.

If you start to make a patch and change your mind, use the Zmacs Cancel Patch command. This command deletes the record stating that this patch is being worked on. It also tells the editor that you are no longer editing a patch.

Saving to Disk

23.11 Of all the procedures described in this section, the most common one is to take a partition containing a Lisp load band, update it to have all the latest patches, and save it into a partition. The **load-and-save-patches** function does all this conveniently for you.

If you want to do something other than loading only the latest patches, you must perform the steps by hand. Start by cold booting the machine to get a fresh system. Next, you must log in with no INIT file (so that when you save the Lisp image, the side effects of the INIT file are not saved, too). Now you can load in any new software you want. Usually, you should also execute a **load-patches** function to update all the loaded systems. You may also want to call **sys:set-system-status** to change the release status of the system.

When you have finished loading everything, execute a **print-disk-label** function to find a partition in which to save your new Lisp environment. It is recommended that you do not reuse the current partition. Though this operation works, it is somewhat slower, and if an error occurs while the partition is being saved (for instance, after half of the current partition is written), it will probably be impossible to cold boot the current partition again. Once you have found the partition, use the **disk-save** function to save everything into that partition.

load-and-save-patches &optional *partition unit* {*option*}* Function

This function loads patches and saves a band, with a simple user interface. Run this function immediately after cold booting, without logging in first; it logs in as LISPM. Any *options* are passed as arguments to **load-patches**. After loading the patches, it prints the disk label and asks you for the band in which to save the current world, unless you have specified *partition*. *unit* should identify the disk unit number that *partition* is on; the default is the system default unit number. Finally, it saves the band as described in **disk-save**.

disk-save *partition* &optional *unit* &key *no-query partition-comment display mode* Function

This function saves the current Lisp world in the designated partition and unit, which defaults to the system default unit number. The *partition* argument can be a partition name (a string), or it can be a number *n*, in which case the name LOD*n* is used. The following keyword options exist for **disk-save**:

:no-query — If this keyword is true, no interactive confirmation is solicited from the keyboard. The default is to ask the user.

:partition-comment — This keyword is a string that describes the new Lisp environment to be put in the disk label. This is normally prompted for, so is of use only when **:no-query** is true.

:display-mode — This keyword controls the type of **disk-save**'s status display. The values are **normal** (the default screen display) and **nil** (no display).

This function first tries to determine if the current environment will fit in the specified load band. If not, a message is printed and **disk-save** exits. You can determine the current environment size yourself by using **sys:estimate-dump-size**.

This function first asks you for yes-or-no confirmation to indicate if you actually want to write over the named partition. Then it tries to determine what to put into the textual description of the label. It starts with the brief version of **sys:system-version-info** (described earlier in this section). Then it asks you for an additional comment to append to this; usually, you press RETURN at this point, but you can also add a comment that is returned by **sys:system-version-info** (and thus printed when the system is booted) from then on. If this comment does not fit into the fixed size available for the textual description, the function asks you to retype the information (version information as well as your comment) in a compressed form that fits. The compressed version appears in the textual description in **print-disk-label**.

The Lisp environment is then saved into the designated partition, and the equivalent of a cold boot is performed from this partition.

Once the patched system has been successfully saved and the system comes back up, you can make it current with **set-current-band**.

NOTE: You may not want to save patched systems after running the editor or the compiler. Although this procedure works and makes the editor or compiler start up more quickly in the saved band, it makes the saved system considerably larger. To produce a clean saved environment, you should try to do as little as possible between the time you cold boot and the time you save the partition. Additionally, you should perform a garbage collection prior to saving a patched system in order to compress the environment (see Section 25, Storage Management).

---

print-login-history &optional *stream history* Function

This function prints out historical information contained in *history*. The information is printed on *stream*, which defaults to ***standard-output***. The default value for *history* is **sys:login-history**, which includes data such as who was logged in when previous **disk-saves** were performed, which machine the save was performed from, and the date and time when the save was performed.

sys:login-history Variable

The value of this variable is a list of entries, one for each person who has logged in to this band after it was created. This history makes it possible to identify who executed a **disk-save** on a band containing something broken. Each entry is a list of the user ID, the host used for login, the Explorer system on which the band was being executed, and the date and time.

System Status

23.12 The patch system has the concept of the *status* of a major version of a system. The status is displayed when the system version is displayed, in places such as the system greeting message and the disk partition comment. This status allows users of the system to know what is going on. The system status changes as patches are made to the system.

The status is indicated by one of the following keywords:

:experimental — The system has been built but has not yet been fully debugged and released to users. This is the default status when a new major version is created, unless it is overridden with the **:initial-status** option to **defsystem**.

:released — The system is released for general use. This status produces no extra text in the system greeting and the disk partition comment.

:obsolete — The system is no longer supported.

:broken — This keyword resembles **:experimental** but is used when the system was incorrectly thought to have been debugged and hence was temporarily **:released**.

sys:set-system-status *system status* &optional *major-version* Function

This function changes the status of a system. The *system* argument is the name of the system. The *major-version* argument is the number of the major version to be changed. The patch directory for that version should already exist. If unsupplied, it defaults to the version currently loaded. The *status* argument should be one of the keywords above.

Common Lisp Modules

23.13 The following forms are defined by Common Lisp and are used to monitor and control application modules. In this context, modules are distinct from the entities defined in the **defsystem** macro. For Common Lisp, modules roughly correspond to a *system* definition. In this sense, ***modules*** provides a list of loaded software and the **provide** and **require** functions can be used to control loading the files that make up and keep track of the environment. Note that module names are case sensitive.

modules *[c]* Variable

This variable keeps track of the modules currently loaded in the Lisp environment by maintaining a list of the module names. The **provide** and **require** functions use this variable.

provide *module-name* *[c]* Function

This function appends *module-name* to the list of currently loaded modules contained in ***modules***.

require *module-name* &optional *pathname* *[c]* Function

This function first checks to see if *module-name* is contained in ***modules***. If not, this function loads the module. If *pathname* is supplied, it specifies the file or files that make up the module to be loaded. The *pathname* argument specifies either a single pathname or a list of pathnames. If more than one file is specified, they should be listed in the order in which they are to be loaded. If *pathname* is unsupplied or is **nil**, the system tries to figure out which files should be loaded by assuming that *module-name* is a defined system. If *module-name* is a system name, then a **make-system** is performed with a **:noconfirm**.

For both of these functions, *module-name* should be a string or a symbol. If it is a symbol, the symbol's print name serves as the module name. If the module is made up of only one package, the module name is often the same as the package name.

Simple System Maintenance

23.14 The following set of functions may be useful for performing simple system maintenance.

sys:load-if *pathname* &key **:package :verbose :set-default-pathname :if-does-not-exist :print** Function

This function loads *pathname* if it needs to be loaded; that is, if *pathname* has been changed since it was last loaded or has never been loaded, it is loaded by this function. The keyword arguments are passed to **load** if *pathname* is to be loaded. For more details on the remaining arguments, see the description of **load**.

sys:compile-if *pathname* &key **:force-date :output-file :load :verbose :set-default-pathname :package :declare :suppress-debug-info** Function

This function compiles *pathname* if it needs to be compiled; that is, if the *pathname* source has a newer version than its binary counterpart, a **compile-file** is executed on *pathname*. If **:force-date** is non-**nil**, then the creation date is used instead of the version number to determine if the file needs to be recompiled. If **:force-date** is **nil**, then the value of **compiler:*output-version-behavior*** determines whether *pathname* is compiled. For more details on the remaining keyword arguments, see the description of **compile-file**.

sys:compile-load-if *pathname* &key **:force-date :output-file :verbose :set-default-pathname :package :declare :suppress-debug-info :if-does-not-exist :print** Function

This function compiles and/or loads *pathname* if necessary. If **:force-date** is non-**nil**, then the creation date is used instead of the version number to determine if the file needs to be recompiled. For more details on the remaining keyword arguments, see the description of **compile-file** and **load**.

sys:dep-compile-if *pathname dep-pathnames* &key **:force-date :output-file :load :verbose :set-default-pathname :package :declare :suppress-debug-info** Function

This function compiles *pathname* if necessary or if any of *dep-pathnames* are newer than the binary version of *pathname*. If **:force-date** is non-**nil**, then the creation date is used instead of the version number to determine if the file needs to be recompiled. If **:force-date** is **nil**, then the value of **compiler:*output-version-behavior*** determines whether *pathname* is compiled. For more details on the remaining keyword arguments, see the description of **compile-file**.

sys:dep-compile-load-if *pathname dep-pathnames* &key **:force-date :output-file :verbose :set-default-pathname :package :declare :suppress-debug-info :if-does-not-exist :print** Function

This function compiles *pathname* if necessary or if any of *dep-pathnames* are newer than *pathname*; then this function loads *pathname* if necessary. If **:force-date** is **nil**, then the value of **compiler:*output-version-behavior*** determines whether *pathname* is compiled and/or loaded. For more details on the remaining keyword arguments, see the descriptions of **compile-file** and **load**.

DATES AND TIMES

24

Introduction

24.1 The TIME package contains a set of functions for manipulating dates and times: finding the current time, reading and printing dates and times, converting between formats, and other miscellany regarding peculiarities of the calendar system. It also includes functions for accessing the Explorer microsecond timer.

Times are represented in two different formats by the functions in the TIME package. One way is to represent a time by several numbers, indicating a year, a month, a date, an hour, a minute, and a second (as well as, sometimes, a day of the week and a timezone). This is called the *decoded* format. If a year less than 100 is specified, a multiple of 100 is added to it to bring it within 50 years of the present. Year numbers returned by the time functions are greater than 1900. The month is 1 for January, 2 for February, and so on. The date is 1 for the first day of the month. The hour is a number from 0 to 23. The minute and second are numbers from 0 to 59. Days of the week are fixnums, where 0 means Monday, 1 means Tuesday, and so on. A timezone is specified as the number of hours west of Greenwich Mean Time (GMT); thus, in Massachusetts the timezone is 5, and in Texas the timezone is 6. Any adjustment for daylight savings time is separate from this. However, daylight savings time is considered to be in effect if the date in question is between 2:00 am on the last Sunday in April and 1:00 am on the last Sunday in October.

The *decoded* format is convenient for printing out times in a readable notation, but it is inconvenient for programs to make sense of these numbers and pass them around as arguments (because there are so many of them). Thus, there is a second representation called Universal Time, which measures a time as the number of seconds since January 1, 1900, at midnight GMT. This *encoded* format is easy to deal with inside programs, although it does not make much sense to look at (it looks like a huge integer). Consequently, both formats are provided; there are functions to convert between the two formats; and many functions exist in two versions, one for each format.

The Explorer hardware includes a timer that counts once every microsecond. It is controlled by a crystal and thus is fairly accurate. The absolute value of this timer does not mean anything useful because it is initialized randomly. You read the timer at the beginning and end of an interval and then subtract the two values to get the length of the interval in microseconds. These relative times allow you to measure intervals of up to 71 1/2 minutes (32 bits) with microsecond accuracy.

The Explorer system keeps track of the time of day by maintaining a *timebase*, using the microsecond clock to count off the seconds. You can set the timebase by using **time:set-local-time**, described in paragraph 24.2, Getting and Setting the Time.

A similar timer that counts in 60ths of a second rather than in microseconds is useful for measuring intervals of a few seconds or minutes with less accuracy. Schedules of periodic housekeeping functions of the system are based on this timer.

Getting and Setting the Time

24.2 The following functions can be used to get and set the time.

get-decoded-time *[c]* Function

This function gets the current time in decoded form. It returns nine values: seconds, minutes, hours, date, month, year, day-of-the-week, the value returned by **time:daylight-savings-time-p**, and timezone, with the same meanings as for **decode-universal-time** (see paragraph 24.7, Time Conversions). If the current time is not known, **nil** is returned.

get-universal-time *[c]* Function

This function returns the current time in Universal Time.

time:set-local-time &optional *new-time* Function

This function sets the local time to *new-time*, which must be either a Universal Time or a suitable argument to **time:parse-universal-time** (see paragraph 24.6, Reading and Printing Time Intervals). If *new-time* is **nil** (the default), this function checks to see if the local system clock can be used. If not, **time:set-local-time** prompts the user to supply the time. Note that normally you do not need to call this function because the booting procedure of the Explorer can usually determine the time automatically. This function is useful mainly when the timebase does not function properly for one reason or another.

Elapsed Time

24.3 The following functions do not deal with calendar dates and times but with elapsed time in 60ths of a second. These times are used for many internal purposes where the idea is to measure a small interval accurately, not to depend on the time of day or day of month.

time &optional *form* *[c]* Macro

With no argument, this function returns a number that increases by one every 60th of a second. The value wraps around roughly once a day. Use the **time-lessp** and **time-difference** functions to avoid possible trouble due to the wraparound.

When given an argument, **time** evaluates this argument and informs the ***trace-output*** stream of how long the evaluation took. The returned value is the result of evaluating *form*. For more details, see the Performance Tools section of the *Explorer Tools and Utilities* manual.

get-internal-run-time *[c]* Function
get-internal-real-time *[c]* Function

These functions return the total time in 60ths of a second since the last boot. This value does not wrap around. Eventually it becomes a bignum. The Explorer system does not distinguish between run time and real time.

time:microsecond-time Function

This function returns the value of the microsecond timer, as a bignum. The values returned by this function wrap around back to 0 about once per hour.

internal-time-units-per-second *[c]* Constant

According to Common Lisp, this is the ratio between a second and the time used by values of **get-internal-real-time**. On the Explorer, this value is 60. This value may be different in other Common Lisp implementations.

time-lessp *time1 time2* Function

This function returns true if *time1* is earlier than *time2*, compensating for wraparound. Otherwise, this function returns **nil**.

time-difference *time1 time2* Function

Assuming that *time1* is later than *time2*, this function returns the number of 60ths of a second difference between them, compensating for wraparound.

time-increment *time interval* Function

This function increments *time* by *interval*, wrapping around if appropriate. Both arguments are measured in 60ths of a second.

Printing Dates and Times

24.4 The functions in this section create printed representations of times and dates in various formats and then send the characters to a stream. If you pass **nil** to any of these functions as the *stream* parameter, the functions return a string containing a printed representation of the time, instead of printing the characters to any stream.

time:*default-date-print-mode* Variable

This variable holds the default for the *date-print-mode* argument to each of the functions described in this numbered paragraph. Initially, the value of this variable is **:mm/dd/yy**.

The three functions **time:print-time**, **time:print-universal-time**, and **time:print-current-time** accept an argument called *date-print-mode* whose purpose is to control how the date is printed. It always defaults to the value of **time:*default-date-print-mode***. Possible values include the following:

:dd/mm/yy	; Print the date as in 16/3/53
:mm/dd/yy	; Print the date as in 3/16/53
:dd-mm-yy	; Print the date as in 16-3-53
:dd-mmm-yy	; Print the date as in 16-Mar-53
:|dd mmm yy|	; Print the date as in 16 Mar 53
:ddmmmyy	; Print the date as in 16Mar53
:yymmdd	; Print the date as in 530316
:yymmmdd	; Print the date as in 53Mar16

time:print-current-time &optional *stream date-print-mode* Function

This function prints the current time formatted according to the argument *date-print-mode*. The default value for *date-print-mode* is the current value of **time:*default-date-print-mode***. If *stream* is **nil**, this function returns the formatted string as its value; the default is ***standard-output***.

time:print-time *seconds minutes hours date month year* &optional *stream date-print-mode* — Function

This function prints the specified time formatted according to the argument *date-print-mode*. The default value for *date-print-mode* is the value of **time:*default-date-print-mode***. If *stream* is **nil**, this function returns the formatted string as its value; the default is ***standard-output***.

time:print-universal-time *universal-time* &optional *stream timezone date-print-mode* — Function

This function prints the specified time formatted according to the argument *date-print-mode*. The default value for *date-print-mode* is the value of **time:*default-date-print-mode***. If *stream* is **nil**, this function returns the formatted string as its value; the default is ***standard-output***. The *timezone* argument defaults to the value of **time:*timezone***.

time:print-brief-universal-time *universal-time* &optional *stream reference-time date-print-mode* — Function

This function is like **time:print-universal-time** except that it omits seconds and prints only those parts of *universal-time* that differ from *reference-time*, a universal time that defaults to the current time. Thus, the output looks like one of the following forms:

```
02:59                ; the same day
3/4 14:01            ; a different day in the same year
8/17/74 15:30        ; a different year
```

The date portion may be printed differently according to the argument *date-print-mode*. The *stream* argument defaults to the value of ***standard-output***.

time:print-current-date &optional *stream* — Function

This function prints the current date, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream. The *stream* argument defaults to the value of ***standard-output***.

time:print-date *seconds minutes hours date month year day-of-the-week* &optional *stream* — Function

This function prints the specified date, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream. The *stream* argument defaults to the value of ***standard-output***.

time:print-universal-date *universal-time* &optional *stream timezone* — Function

This function prints the specified time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream. The *timezone* argument defaults to the value of **time:*timezone***.

Reading Dates and Times

24.5 The functions discussed here accept most reasonable printed representations of date and time and then convert them to the standard internal forms. The following are representative formats accepted by the parser.

```
"March 15, 1960"        "3/15/60"                "3/15/1960"
"15 March 1960"         "15/3/60"                "15/3/1960"
"March-15-60"           "3-15-60"                "3-15-1960"
"15-March-60"           "15-3-60"                "15-3-1960"
"15-Mar-60"             "3-15"                   "15 March 60"
"Fifteen March 60"      "The Fifteenth of March, 1960"
"Friday, March 15, 1960"
```

```
"1130."         "11:30"          "11:30:17"      "11:30 pm"
"11:30 AM"      "1130"           "113000"
"11.30"         "11.30.00"       "11.3"          "11 pm"

"12 noon"       "midnight"       "m"   "6:00 gmt"    "3:00 pdt"
```

Any date format may be used with any time format.

```
"Two days after March 3, 1960"
"Three minutes after 23:59:59 Dec 31, 1959"

"Now"    "Today"    "Yesterday"      "five days ago"
"two days after tomorrow"            "the day after tomorrow"
"one day before yesterday"
```

The following means one minute after midnight:

```
"One minute after March 3, 1960"
```

time:parse *string* &optional *start end futurep base-time must-have-time date-must-have-year time-must-have-second day-must-be-valid* Function

This function interprets *string* as a date and/or time, and returns nine values: seconds, minutes, hours, date, month, year, day-of-the-week, the value returned by **time:daylight-savings-p**, and relative-p. The returned value for relative-p is true if the string includes a relative part, such as `one minute after` or `two days before` or `tomorrow` or `now`; otherwise, it is **nil**. The arguments *start* and *end* delimit a substring of *string*. If *end* is **nil**, the end of the string is used. The argument *must-have-time* means that *string* must not be empty. The argument *date-must-have-year* means that a year must be explicitly specified. The argument *time-must-have-second* means that the second must be specified. The argument *day-must-be-valid* means that if a day of the week is given, then it must actually be the day that corresponds to the date. The argument *base-time* provides the defaults for unspecified components; if it is **nil**, the current time is used. The argument *futurep* means that the time should be interpreted as being in the future; for example, if the base is 5:00 and the string refers to the time 3:00, then the time refers to the next day if *futurep* is non-**nil**.

If the input is not valid, the error condition **sys:parse-error** is signaled.

The *start* argument defaults to 0; the *end* argument defaults to **nil**; the *futurep* argument defaults to **t**; and the *day-must-be-valid* argument defaults to **t**.

time:parse-universal-time *string* &optional *start end futurep base-time must-have-time date-must-have-year time-must-have-second day-must-be-valid* Function

This function is the same as **time:parse** except that it returns two values: an integer, representing the time in Universal Time, and the relative-p value.

The *start* argument defaults to 0; the *end* argument defaults to **nil**; the *futurep* argument defaults to **t**; and the *day-must-be-valid* argument defaults to **t**.

Reading and Printing Time Intervals

24.6 In addition to the functions for reading and printing instants of time, there are other functions specifically for printing time intervals. A time interval is either a number (measured in seconds) or **nil**, meaning *never*. The printed representations for actual intervals have the format `3 minutes 23 seconds`, whereas the format for **nil** is `Never` (some other synonyms and abbreviations for *never* are accepted as input).

time:print-interval-or-never *interval* &optional *stream* Function

This function writes onto *stream* the printed representation for *interval* as a time interval. The *interval* argument should be a nonnegative fixnum (indicating seconds) or **nil**. The *stream* argument defaults to the value of ***standard-output***.

time:parse-interval-or-never *string* &optional *start end* Function

This function converts *string*, a printed representation for a time interval, into a number (indicating seconds) or **nil**. The arguments *start* and *end* can be used to specify a portion of *string* to be used; the default is to use all of *string*. An error is signaled if the contents of *string* do not look like a reasonable time interval. The following are some examples of acceptable strings:

```
"4 seconds"            "4 secs"              "4 s"
"5 mins 23 secs"       "5 m 23 s"            "23 SECONDS 5 M"
"3 yrs 1 week 1 hr 2 mins 1 sec"
" "                    "not ever"            "no"           "never"
```

Note that several abbreviations are understood, the components can be in any order, and case (uppercase versus lowercase) is ignored. Also, months are not recognized, because various months have different lengths and there is no way to know which month is being referred to. This function always accepts anything that was produced by **time:print-interval-or-never**; furthermore, it returns exactly the same integer (or **nil**) that was printed.

time:read-interval-or-never &optional *stream* Function

This function reads a line of input from *stream* (using **readline**) and then calls **time:parse-interval-or-never** on the resulting string. The *stream* argument defaults to the value of ***standard-output***.

Time Conversions

24.7 The following functions are for converting between time formats.

decode-universal-time *universal-time* &optional *timezone* *[c]* Function

This function converts *universal-time* into its decoded representation. The following nine values are returned: seconds, minutes, hours, date, month, year, day-of-the-week, **time:daylight-savings-time-p**, and the timezone used. The **time:daylight-savings-time-p** value tells you whether daylight savings time is in effect. If so, the hour value is adjusted accordingly. You can specify the timezone value explicitly if you want to know the equivalent representation for this time in other parts of the world. The *timezone* argument defaults to the value of **time:*timezone***.

encode-universal-time *seconds minutes hours date month year* &optional *timezone* *[c]* Function

This function converts the decoded time into Universal Time format and returns the Universal Time as an integer. If you do not specify *timezone,* it defaults to the current timezone, adjusted for daylight savings time. If you do specify *timezone,* it is not adjusted for daylight savings time. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:*timezone* Variable

The value of this variable is the timezone in which this Explorer resides, expressed in terms of the number of hours west of GMT this timezone is. This value does not change to reflect daylight savings time.

Internal Functions

24.8 The following functions provide support for those described previously. Some user programs may need to call them directly, so they are documented here.

time:initialize-timebase Function

This function initializes the timebase by querying time servers to find out the current time. This function is called automatically during system initialization. You can call it yourself to correct the time if it appears to be inaccurate. This function searches for the time from the following sources in this order: the network time server, the system clock, the user. See also **time:set-local-time** in paragraph 24.2, Getting and Setting the Time.

time:daylight-savings-time-p *hours date month year* Function

This function returns true if daylight savings time is in effect for the specified hour; otherwise, it returns **nil**. If the year is less than 100, then 1900 is added to *year*.

time:daylight-savings-p Function

This function returns true if daylight savings time is in effect; otherwise, it returns **nil**.

time:month-length *month year* Function

This function returns the number of days in the specified *month*; you must supply a *year* in case the month is February (which has a different length during leap years). If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:leap-year-p *year* Function

This function returns true if *year* is a leap year; otherwise, it returns **nil**. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:verify-date *date month year day-of-the-week* Function

If the day of the week of the date specified by *date*, *month*, and *year* is the same as *day-of-the-week*, this function returns **nil**; otherwise, it returns a string that contains a suitable error message. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present. The *day-of-the-week* argument is a number between 0 and 6 that represents the days Monday through Sunday, respectively.

time:day-of-the-week-string *day-of-the-week* &optional *mode* Function

This function returns a string representing the day of the week. As usual, 0 means Monday, 1 means Tuesday, and so on. Possible values for *mode* are as follows:

:long	Returns the full English name, such as Monday, Tuesday, and so on. This is the default.
:short	Returns a three-letter abbreviation, such as Mon, Tue, and so on.
:medium	Returns a longer abbreviation, such as Tues and Thurs.
:french	Returns the French name, such as Lundi, Mardi, and so on.
:german	Returns the German name, such as Montag, Dienstag, and so on.
:italian	Returns the Italian name, such as Lunedi, Martedi, and so on.

time:month-string *month* &optional *mode* Function

This function returns a string representing the month of the year. As usual, 1 means January, 2 means February, and so on. Possible values for *mode* are as follows:

:long	Returns the full English name, such as January, February, and so on. This is the default.
:short	Returns a three-letter abbreviation, such as Jan, Feb, and so on.
:medium	Returns a longer abbreviation, such as Sept, Novem, and Decem.
:roman	Returns the Roman numeral for *month* (this convention is used in Europe).

:french	Returns the French name, such as Janvier, Fevrier, and so on.
:german	Returns the German name, such as Januar, Februar, and so on.
:italian	Returns the Italian name, such as Gennaio, Feboraio, and on on.

time:timezone-string &optional *timezone daylight-savings-p* Function

This function returns the three-letter abbreviation for this timezone. For example, if *timezone* is 5, then either EST (Eastern Standard Time) or CDT (Central Daylight Time) is used, depending on the value of *daylight-savings-p*. The *timezone* argument defaults to the value of **time:*timezone***, and the *daylight-savings-p* argument defaults to the value of **time:daylight-savings-p**.

25

STORAGE MANAGEMENT

Storage Management Definitions

25.1 The Explorer system generally has automatic storage management. Storage is allocated when an object is created and can be freed by the garbage collector for reuse when the object is no longer needed. The details of this process do not concern most users. However, at times you can increase the efficiency of storage management, thereby tuning your system's performance by using some of the facilities discussed in this section.

Storage management for Lisp objects is implemented on top of a large, uniform address space provided by the virtual memory management system. The *Lisp Object Space* maps the collection of Lisp objects to the virtual address space. Both the storage allocation system and the garbage collector manage the Lisp Object Space. A facility known as *memory management* deals with the explicit allocation and deallocation of objects.

Virtual Memory Management

25.2 The Explorer system uses *virtual memory*. Virtual memory is a means by which a fast, but comparably smaller, primary store is combined with a larger, but slower, secondary store (up to 128 megabytes for the Explorer system). As with many systems, the Explorer uses semiconductor memory (internal memory) as a primary store and a disk for secondary store. With this arrangement, the Explorer system gains a reasonable simulation of a single, extremely large and fast primary store.

In the Explorer hierarchy, paging is below everything except interrupts; that is, all the system software, except interrupts, depends on paging.

Fixed-size blocks, called *pages*, are moved between primary and secondary stores according to a page management strategy incorporated in the Explorer's system code. The page management strategy moves a page from disk into internal memory whenever an object that resides on that page is referenced. This process is known as *demand paging*. Usually, a page being moved into internal memory must displace another page.

A page replacement policy decides which page must be removed. The Explorer system uses a *page aging* replacement policy, which replaces the page used least recently. If the page selected for replacement has not been altered while in memory, it is merely overwritten. However, a page that has been altered while in memory is called a *dirty page*, and a dirty page chosen for replacement must first be written to the disk. Because of this procedure, every attempt is made to replace an *undirtied* page to avoid the disk write. The Explorer system divides its secondary storage into the load band and the swap space. The load band contains the Lisp system, and its contents are generally assumed to be static. The swap space contains pages that have been created or that have become dirty during system execution.

Some pages, called *wired pages*, are exempted from paging. Wired pages are used for the following:

- Interrupt handler buffers (because interrupts cannot take a page fault)
- I/O buffers and other pages involved in direct memory access (DMA) transfers
- Pages containing paging tables (on which a page fault cannot be allowed)
- Pages containing critical data (that must be accessed without a page fault)

Paging Functions

25.3 The following functions are used to maintain the virtual memory paging system.

NOTE: All memory that can be wired down by some of these functions must be in static areas so that the objects in the memory are not moved by the garbage collector. For more details on static areas, see the description of **make-area**.

---

sys:wire-array *array* &optional *from to* Function

This function wires down *array* preventing it from being paged out. The *from* and *to* arguments are array index lists which can be used to specify a portion of the array to wire down.

sys:unwire-array *array* &optional *from to* Function

This function allows *array* to be paged out within the portion specified by the array index lists *to* and *from*.

sys:wire *obj* Function

This function wires *obj* preventing it from being paged out.

sys:unwire *obj* Function

This function unwires *obj* and allows its underlying memory to be paged out.

sys:page-in-structure *object* Function

This function ensures that the storage that represents *object* is in memory. Any pages that have been swapped out to disk are read in to main memory.

sys:page-in-array *array* &optional *from to* Function

This function is a version of the **sys:page-in-structure** function that can bring in a portion of an array. The *from* and *to* arguments are lists of subscripts. If they are shorter than the dimensions specified for *array*, the remaining subscripts are assumed to be zero.

sys:page-in-area *area-number* Function
sys:page-in-region *region-number* Function

These functions bring into memory all the swapped out pages of the specified *area-number* or *region-number*.

sys:set-disk-switches &key **:clean-page-search :time-page-faults :multi-page-swapouts :multi-swapout-page-count-limit :serial-delay-constant** Function

This function allows you to set parameters to modify the various paging parameters.

:clean-page-search — When the value for this keyword is 0, the page replacement algorithm scans through physical memory looking for a clean page to flush while looking for physical memory. The default value for this keyword is 1, which turns the search on.

:time-page-faults — When the value for this keyword is 1, **%total-page-fault-time** is enabled in the counter block. The value of the counter is the time (in microseconds) spent in the page fault microcode plus the disk wait time, but excluding code that resolves page exceptions. The default value for this option is 0, which disables this operation.

:multi-page-swapouts — When the value for this keyword is 1 (the default), the page replacement algorithm cleans adjacent memory page images by writing them to disk in the same disk write for a page being flushed.

:multi-swapout-page-count-limit — This option specifies the maximum number of pages that can be updated in a multi-swapout. The value for this options can be any integer between 0 and 255. The default value is 128.

:serial-delay-constant — This option specifies the timing constant for microcode access to the serial chip registers. This value must *not* be less than 12 (the default), which produces a delay of at least 2.641 microseconds on the Explorer system. This option should be changed with extreme caution.

The following functions are a bit more primitive because they accept a virtual address number, which should be a fixnum. You can obtain these addresses by using **%pointer** and similar subprimitive functions documented in the *Explorer System Software Design Notes*.

sys:wire-page *address* &optional *wire-p* Function

This function *wires-down* (prevents from being paged out) the page containing *address* if *wire-p* is true. If the *wire-p* argument is **nil**, then the pages containing *address* are unwired.

sys:unwire-page *address* Function

This function unwires the page containing *address*. In other words, it does the same thing as sys:wire-page with a **nil** value for the *wire-p* argument.

sys:page-in-words *address n-words* Function

This function reads in any pages that have been swapped out to disk in the range of address space starting at *address* and continuing for *n-words* in the fewest possible disk operations.

Address Space and Swap Space

25.4 Although the *maximum* address space of the Explorer system is 128 megabytes, the *available* address space during execution is determined by the swap space.

One or more partitions on disk supply the swap space. When the system is booted, all online disks are searched for page partitions. There can be more than one page partition. The total available swap space is approximately the sum of the sizes of the page partitions, but no more than 128 megabytes can be used.

The swap space size need not equal the maximum address space size to use that address space. Because portions of the system are never swapped, 128-megabyte swapping store is not needed to get the use of all virtual memory. Although the original load band is logically part of the address space, it does not occupy swap space unless altered during execution. The following factors affect the amount of swap space needed:

- Rate of creation of new objects
- Frequency of garbage collection
- Changes to the base system

The garbage collection daemon process constantly monitors the use of swap space. If swap space is running low, the process issues periodic warning notifications, such as the following:

```
Swap space low. Total blocks: 35000, Free blocks: 350 (10%)
```

If you receive a swap space warning, you should increase your swap space or consider rebooting (that is, save edit buffers, close files, and so forth). To increase your swap space, use **sys:edit-disk-label** to add another page partition and then call **sys:change-swap-space-allocation**. As the remaining space is used up, you receive additional warnings. If you ignore these warnings about low amounts of swap space, the system will eventually run out of swap space and crash. You can check the current swap space usage by invoking the **sys:swap-status** function. Summary information on swap status is also provided in the **gc-status** display.

sys:swap-status &optional *stream* Function

This function writes the status of the current swap space to *stream*, which defaults to the value of ***standard-output***. This function returns three values: the total usable swap size, the number of pages free, and the number of pages used. The following is an example display:

```
Status for Logical Page Device 0.
  On disk unit number 0, a LOD band (read-only)
  Starting block: 42507, size: 33641 pages

Status for Logical Page Device 1.
  On disk unit number 1, a PAGE band (read-write)
  Starting block: 41761, usable size: 34000 pages, used: 1808 pages (5%)

Total Read-Write swap space 34000 pages on 1 swap bands, free 32192 pages (95%)
```

Note that the swap partition sizes are expressed in units of virtual memory pages. One page equals two disk blocks.

Swap space can possibly be increased during execution by adding a page partition (see the **edit-disk-label** function in the *Explorer Input/Output Reference* manual) and then invoking the **change-swap-space-allocation** function.

sys:change-swap-space-allocation Function

Invoking this function reconfigures swap space according to the disk labels of all online disks. You can use this command to increase swap space during execution by adding page bands using the label editor, and then invoking the **sys:change-swap-space-allocation** function to inform the system of the change. Note that you cannot enlarge an existing page band and have this function recognize the change. If you enlarge an existing page band, you must reboot for the system to recognize the new size.

If, for example, you edit the disk label to add a new page band, when you invoke the **sys:change-swap-space-allocation** function, the Explorer system makes the requested changes and, finally, calls **sys:swap-status**.

Storage Allocation and Areas

25.5 To minimize the paging requirements for your application program, the Explorer system provides a way to divide internal memory into *areas*. Each area contains related objects, of any type. By separating frequently used data and rarely used data into different areas, you limit the number of pages required for the frequently used data, thus requiring fewer transfers between disk and internal memory. As a result, your application program runs more quickly and efficiently.

For example, the system puts structures dealing with the debugging information about compiled functions in a special area, thereby compacting other list structures pointed to by Lisp functions.

When a new object is created, your program can specify the area where it is to reside as an option. For example, instead of using **cons** you can use **cons-in-area** (see Section 6, Lists and List Structure).

Object-creating functions that take keyword arguments generally accept an **:area** argument. You can also control which area is used by binding **default-cons-area**, which is discussed later in this section. Most functions that allocate storage will use the value of this variable, by default, to specify which area to use.

There is a default **working-storage-area** that collects those objects you choose not to control explicitly. This is where the majority of user-created objects are created.

Either of the following forms may be used to create an array in the area `my-area`. In the first example, any object creation that is done while executing `<other-forms>` will also take place in `my-area`.

```
(let ((default-cons-area my-area))
     (make-array 500.)
     ... <other forms>)
```

```
(make-array 500. :area my-area)
```

The following functions are also available for explicitly creating objects in areas:

cons-in-area *car cdr area* Function
list-in-area *area* &rest *elements* Function
list*-in-area *area first* &rest *elements* Function

Each area has a name and a number. The name is a symbol whose value is the number. The number is an index into various internal tables. Currently, the maximum number of areas is 256.

An area's storage consists of one or more *regions*. Each region is a contiguous section of address space with certain homogeneous properties, the most important of these being the *data representation type*. A given region can only store one type. The two types that exist now are *list* and *structure*. A list region holds only cons cells and cdr-coded list structures whose components are always fully tagged. Other Lisp objects (which may not always be fully tagged) are allocated in structure space. These objects include arrays, flavor instances, compiled function objects, and large numbers. Because lists and structures cannot be stored in the same region, they cannot be on the same page. This is an important point to remember when you use areas to optimize the locality of reference.

When you create an area, one region is created initially. When you try to allocate memory to hold an object in a particular area, the system tries to find a region that has the right data representation type to hold this object and that has enough room for it to fit. If no such region exists, it makes a new one (or signals an error; see the **:size** option to **make-area**, below). Currently, there is a system limit of 2048 regions.

Areas do not consume address space. Address space (up to 128 MB) is *allocated* to regions as they are created. As objects are created in the area, they *use* the allocated address space in the area's regions. These distinctions are discussed further in the discussion of **describe-area** and **describe-region**.

Area Functions and Variables

25.6 The following functions and variables are used in conjunction with areas.

default-cons-area Variable

The value of this variable is the number identifying the area in which objects are created by default. It is initially the **working-storage-area**. When you specify **nil** in response to an argument requiring an area, that argument then uses the value of **default-cons-area**. Note that to put objects into an area other than **working-storage-area**, you can either bind this variable or use functions such as **cons-in-area**, which take the area as an explicit argument.

sys:**background-cons-area** Variable

The value of this variable is the number identifying a nontemporary area in which objects should be created if they are incidental side effects from a system function. This area is used whenever an object is created that should never be in a temporary area, even if **default-cons-area** is a temporary area.

By default, this area is **working-storage-area**.

sys:%address-space-quantum-size Constant

This constant, whose current value is 16,384 words (32 pages), is the increment in which address space is assigned to regions. A region will have no fewer than this number of words and will have a multiple of this quantum.

make-area &key **:name :size :region-size :representation :gc :read-only :pdl :room** Function

This function creates a new area, whose name and attributes are specified by the keywords. You must specify a symbol as a name; the symbol is set to the area number of the new area, and this number is also returned so that you can use **make-area** as the initialization of a **defvar**. The following keywords can be used with this function:

:name — The value of this keyword is a symbol that names the area. The area number that is created is assigned as the value of the symbol. This argument is required.

:size — The value of this keyword specifies the maximum allowable size of the area, in words. If the number of words allocated to the area reaches this size, attempting to cons an object in the area signals an error. The default value for an area's size is a special flag indicating that the area should be allowed to grow arbitrarily large without an error being signaled.

:region-size — The value of this keyword specifies the approximate size, in words, for old regions within this area. The default is 4 address space quanta, which is equal to 64,000 words. This option should always be specified in increments equal to the **sys:%address-space-quantum-size** constant. If an area expands to the point where it requires a new region, the Explorer system will generally create the smallest possible region (one address space quantum) to hold the object. If objects in this area become old and survive several garbage collections, they are placed in regions of this default size, if possible.

NOTE: If you specify **:size** and not **:region-size**, the area will have exactly one region, making all the area's virtual address space contiguous and making the area unexpandable.

:representation — The value of this keyword identifies the type of object to be contained in the area's initial region. The argument to this keyword can be **:list** or **:structure**. The **:structure** argument is the default.

:gc — This keyword controls how garbage collection affects the area. The choices are the following:

- **:dynamic** — Objects in dynamic areas can be moved by the garbage collection process. This is the default value. If objects in this area are to be wired down, the area should be created as a **:static** area.

- **:static** — Objects in static areas cannot be moved (or collected) by garbage collection. Use this gc-type for areas that will contain I/O buffers that must be wired down. Because garbage in static areas cannot be collected, you should only specify **:static** when absolutely necessary, that is, when the area is to contain wired down objects.

- **:temporary** — Temporary areas are now synonymous with **:dynamic** areas.

:read-only — With an argument of true, this keyword limits the area to read-only. The value of **:read-only** defaults to **nil**. If an area is read-only, then any attempt to change anything in it (altering a data object in the area or creating a new object in the area) signals an error unless **sys:%inhibit-read-only** is bound to a non-**nil** value. For more information, see the *Explorer System Software Design Notes*).

:room — With an argument of true, this keyword adds this area to the list of areas that the **room** function displays by default.

Consider the following example:

```
(make-area :name 'foo-area
      :gc :dynamic
      :region-size (* 4 sys:%address-space-quantum-size)
      :representation :list)
```

This form creates an area named `foo-area`, which is expandable and is intended to contain a mixture of objects with different lifespans (some short, others perhaps permanent). The area's old regions will be 64 pages each (32,000 words), and the initial region is of type `list`.

area-list Constant

The value of this variable is a list of the names of all existing areas. This list shares storage with the internal area name table, so do not change it.

area-name *number* Function

Given an area number, this function returns the name as a symbol. The value for *number* cannot be larger than 255. If there is not an area that corresponds to *number*, **nil** is returned.

---

describe-area *area* &key **:base :verbose** Function
describe-region *region* &key **:base** Function

These functions provide information on the current state of memory allocation for a particular *area* or *region*. If **:base** is supplied, then the virtual addresses that are printed are formatted in that base; the default is 10.

For **describe-area**, if **:verbose** is true (the default), then **describe-region** is called, with its default arguments, for each region in the specified area.

Consider the following examples:

```
(make-area :name 'a-dynamic-area)               ; Make an area.
(make-array 100 :area a-dynamic-area)           ; Something in structure region.
(make-list 100 :area a-dynamic-area)            ; Something in list region.
(describe-area a-dynamic-area :base 10.))
```

The following is then printed on the screen:

```
Area 71: A-DYNAMIC-AREA
There are now 32768 words assigned, 201 used. The area is growable.
Region size 65536
Default cons generation = 0
It currently has 2 regions.
251: 22331392 Origin, 16384 Length, 100 Used, 0 GC, Type LIST  NEW, Gen 0
250: 22265856 Origin, 16384 Length, 101 Used, 0 GC, Type STRUC NEW, Gen 0
```

In this example, a new area is made and then something is put in each type of region. This display is done in base 10, and the default is to print information on each region. Note that the region numbers are 250 and 251. For each region, the following definitions are used:

- Origin — The virtual address of the origin of this region.
- Length — The total allocation of this region in words.
- Used — The number of words in this region currently used by objects.
- GC — A scavenger pointer; only meaningful when garbage collection is active in this region.
- Type — The type of data in this region; the type is either LIST or STRUCTURE.
- Space — The current space type, which is one of the following:
    - NEW — New objects are being created in this region.
    - OLD — Old space which is being collected.
    - COPY — Objects are being copied to this region from old space by the garbage collector.
    - STATIC — This region contains very old objects or objects that must not be moved by the garbage collector.
    - FIXED — Similar to STATIC; for system use only.
- Gen — The age of objects in this region; this value is 0, 1, 2, or 3, where 3 is the oldest.
- *flags* — The following informative flags can appear at the end of this line:
    - READ-ONLY — This region is read-only.
    - MAR — The memory address register points to something in this region.

Now consider the following example of **describe-region**:

```
(describe-region 251.:base 8.)
```

The following is then printed on the screen:

```
251: #o125140000 Origin, #o200000 Length, #o144 Used, #o0 GC, Type LIST NEW, Gen 3
```

The information for this region is the same as that in the preceding **describe-area** example except that the information for **describe-region** is printed in base 8. The default base is base 10.

room &rest *areas* *[c]* Function
room Variable

The **room** function prints to ***standard-output*** the allocated size and used size of the specified *areas*. If *areas* is **t**, then all areas are shown. If *areas* is not supplied, then only those areas whose names are included in the value of the **room** variable are used. If *areas* is specified as **nil**, then only the header for the display is printed. Consider the following example:

```
(room nil)
```

The following is then printed on the screen:

```
Physical Memory: 2,097,152 words (8 MB). Wired Pages: 114 System + 38 User.
Address space free size: 4,142,080 words (8,090 pages).
```

This is the basic header. It tells you the amount of physical memory (8 MB), the number of wired pages (142 in all), and the amount of free address space. Now consider another example:

```
(make-area :name 'a-static-area :gc :static :room t :size
(* 2 sys:%address-space-quantum-size))
(make-array 400 :area a-static-area)
(room)
```

The following is then printed on the screen:

```
Physical Memory: 2,097,152 words (8 MB). Wired Pages: 114 System + 38 User.
Address space free size: 922,624 words (1,802 pages).
A-STATIC-AREA........... 1 region           400/16384 used (  2%)
WORKING-STORAGE-AREA.....33 regions   2858847/3244032 used (growable)
MACRO-COMPILED-PROGRAM...14 regions   1034060/1114112 used (growable)
```

This example shows the header and information on the defaulted areas specified in the **room** variable. For each area, it states the number of regions, the ratio of space being used to the space allocated (in words), and an indication if the area can be expanded. If it cannot be expanded, a decimal percentage of the previous ratio is given. Recall that you can add areas to this default display by using the **:room** option with the **make-area** function.

Interesting Areas

25.7 Additional areas of interest to the user are described below.

working-storage-area Constant

This variable is the normal value of **default-cons-area**. Most working data are consed in this area.

permanent-storage-area Constant

This area is used for permanent system data, which normally never become garbage.

sys:p-n-string Constant

Print names of symbols are stored in this area.

sys:*compiler-symbol-area* Constant
sys:*kernel-symbol-area* Constant
sys:nr-sym Constant
sys:*user-symbol-area* Constant

These areas contain most of the symbols in the Lisp environment, except **t** and **nil**, which are in a different place for historical reasons.

sys:pkg-area Constant

This area contains packages, principally the hash tables with which **intern** keeps track of symbols.

macro-compiled-program Constant

Compiled functions are put in this area by the compiler and by **fasloading** object files.

sys:property-list-area Constant

This area holds the property lists of symbols.

Short Term Objects

25.8 Under some circumstances, it is useful to take more direct control over the time at which an object is deleted from the system. The following forms allow you to control when an object is returned to free space.

with-stack-list (*var* {*expression*}*) {*body-form*}* Special Form
with-stack-list* (*var* {*expression*}* *tail*) {*body-form*}* Special Form

This special form binds *var*, which is not evaluated, to a list that is the evaluation of each *expression*. This list is cdr-coded on the regular PDL; thus, it is temporary and is deleted when the form is exited. As a result, you can change the car but not the cdr of each element in the stack list. The value returned is the value of the last *body-form*. For example:

```
(with-stack-list (foo x y)
   (mumblify foo))
```

The following form is equivalent to the preceding except that `foo`'s value in the first example is a stack list:

```
(let ((foo (list x y)))
   (mumblify foo))
```

The list created by **with-stack-list*** looks like the one created by **list***. The value of *tail* becomes the final cdr rather than an element of the list.

The following is a practical example showing a possible definition for **condition-resume**:

```
(defmacro condition-resume (handler &body body)
   `(with-stack-list* (eh:condition-resume-handlers
                         ,handler eh:condition-resume-handlers)
      . ,body))
```

It is an error to execute **rplacd** on a stack list (except for the tail of one made using **with-stack-list***). However, **rplaca** works normally.

Memory Management Compatibility

25.9 In Release 3.0, the Explorer system no longer uses temporary areas. To be compatible with older code, areas that were made with the **:temporary** option are created as dynamic areas. Other functions that operated on temporary areas exist for compatibility but simply return **nil**. In addition, the **return-storage** facility is no longer supported.

sys:reset-temporary-area *area-number*	Function
return-storage *object*	Function
return-array *object*	Function

These functions exist for compatibility with earlier versions of the Explorer software. They do not perform any action except return **nil**.

sys:page-out-structure *object*	Function
sys:page-out-array *array* &optional *from to*	Function
sys:page-out-pixel-array *array* &optional *from to*	Function
sys:page-out-words *address n-words*	Function
sys:page-out-area *area-number*	Function
sys:page-out-region *region-number*	Function

These functions were available in previous Explorer software releases for notifying the virtual memory system that virtual memory could be reused for other objects. However, in the current virtual memory system these routines have no effect.

Errors Pertaining to Areas

25.10 The following error conditions relate to areas. See Section 20, Error Handling, for details.

sys:area-overflow	Condition

This condition is signaled when you attempt to make an area bigger than its declared maximum size.

The condition instance supports the operations **:area-name** and **:area-maximum-size**. The **sys:area-overflow** condition is based on the **error** flavor.

sys:region-table-overflow	Condition

This condition is signaled if you run out of regions. The **sys:region-table-overflow** condition is based on the **error** flavor.

sys:virtual-memory-overflow Condition

This condition is signaled if all of virtual memory is allocated and an attempt is made to allocate a new region. There may be free space left in some regions in other areas, but there is no way to apply it to the area where storage needs to be allocated. The **sys:virtual-memory-overflow** condition is based on the **error** flavor.

sys:cons-in-fixed-area Condition

This condition is signaled if an attempt is made to add a second region to a fixed area. The fixed areas are certain areas, created at system initialization, that are only allowed a single region because their contents must be contiguous in virtual memory. The **sys:cons-in-fixed-area** condition is based on the **error** flavor.

Garbage Collection

25.11 *Garbage* is dynamically allocated memory that is no longer accessible by any executable code. Garbage is not merely a piece of memory no longer needed by the current program logic. If a piece of memory is truly garbage, then there is no pointer to that memory from any accessible piece of code or in any data accessible by that code—in other words, the system has completely forgotten that this piece of memory exists. *Nongarbage garbage* is a term loosely applied to data objects that the user has forgotten about and the program logic will never use again, but these data objects are still accessible and thus are not true garbage.

Garbage collection (GC) is the process of examining allocated memory to discover which parts of it have become garbage so that they can be made available for reuse. On the Explorer system, garbage collection is performed by copying nongarbage (that is, everything that *can* be accessed) from allocated memory to previously unallocated memory. When the copy operation is complete, all of the previously allocated memory is declared available for reuse. This copying technique ensures that available memory does not become fragmented over time.

The amount of time required to perform any copying garbage collection is proportional to the amount of memory to be searched and proportional to the amount of nongarbage that is to be copied. For example, if essentially all of memory were garbage, then the GC operation would terminate immediately, declaring almost everything to be reusable.

Scavenging is the process of scanning all virtual memory starting from a few well-known anchor points for the purpose of finding all accessible memory. Scavenging guarantees that all potentially accessible memory will be transported from old space to new space during a collection cycle even if the system does not happen to be using it at the time.

Generational Garbage

25.11.1 Characteristic of dynamically allocated memory, most memory that is eventually going to become garbage does so shortly after it is allocated. In other words, the more GC operations a piece of memory survives without becoming garbage, the more likely it is to be permanent data.

Therefore, GC efficiency (garbage reclaimed per unit of time) is greatly improved if the GC is limited to only the most recent generations. A GC of all memory takes a long time, but it eventually finds all of the garbage. A GC limited to *young* memory, on the other hand, tends to find 90 percent of the garbage in 10 percent of the time because it knows the best places to look for garbage. The actual percentages, of course, depend heavily on program characteristics.

Temporal GC

25.11.2 The implementation of generational GC on the Explorer system is called *Temporal GC*, or simply TGC. TGC maintains four generations of memory numbered 0 to 3 (that is, the number represents the number of GC operations the memory contents have survived). Most garbage is located in generations 0 and 1, and memory that has reached generation 3 is virtually static.

Despite the seemingly wide variety of functions available to perform garbage collection and the number of variables available to control these functions, you basically have three sets of choices for TGC:

- Use **batch** or **automatic** (incremental) collection:
  - Batch GC runs in the foreground, takes up most of the machine's resources, and completes the operation as fast as possible.
  - Automatic GC runs in the background, deliberately limits the amount of machine resources it consumes, and never actually finishes the operation because the GC is continuous.
- Determine which generations to collect. This choice is based on how thorough you want to be and how much time you have.
- Decide if you should use promotion. When memory in one generation survives a GC operation, should it be *promoted* to the next higher generation or left where is was?

The automatic TGC algorithm starts collecting generation 0 when it has grown beyond a predefined threshold. Then it may variously perform promoting collections on generations 0 and 1 as their sizes dictate. If an automatic collection of generation 2 is done, however, the survivors are not promoted. Therefore, permanent data slowly migrates into higher generations.

Batch garbage collection functions may be invoked explicitly by the user. They provide keyword arguments by which the generations to collect and the promotion strategy may be controlled.

General GC Functions and Variables

25.11.3 The following functions and variables are used to control GC operations.

gc-status &key *verbose stream* — Function

This function prints various kinds of information about the status of garbage collection to *stream*, whose default is the value of ***standard-output***. GC status information can also be invoked with TERM-G.

gc-immediately &key **:max-gen :promote :silent** Function

This function, invoked with its defaults, performs a relatively quick batch GC, consuming most of the system resources to complete the GC operation as rapidly as possible. This is the appropriate function to use if you want to garbage collect the most garbage-prone areas and then resume exactly where you left off. If automatic GC is on when this function is called, then it is turned off while **gc-immediately** runs and then turned on again when execution has completed.

:max-gen — The number of the oldest generation to be collected. The default is 2, which specifies to collect all generations that are expected to have garbage in them (which excludes generation 3, whose contents are virtually static). Use 3 to collect all possible garbage, but expect it to take quite a long time. There is seldom any reason to use less than 2.

:promote — Controls placement of the data that survives the GC operation in a given generation. If the value for this keyword is true, then data surviving GC in one generation is promoted to the next higher generation. Since promotion into a generation is done before that generation is collected, all nongarbage from all collected generations is placed in the generation above the value of **:max-gen** or in generation 3, whichever is smaller. If the value for **:promote** is false, surviving data is left where it was found. The default for **gc-immediately** value is **nil**.

:silent — When the value of this keyword is true, no notifications are given during the execution of **gc-immediately**. The default is to notify as each generation is collected. A **:silent** value of true also suppresses any user queries that **gc-immediately** might make. For example, if GC detects that there may not be enough space to finish the collection, you are usually warned and provided the option of doing a less extensive collection that requires less space. With a true **:silent** value, GC will do the safest collection without asking you.

full-gc &key **:before-disk-save :duplicate-pnames :max-gen :promote :silent** Function

This function performs a batch GC consuming most of the system resources to complete the GC as quickly as possible. This is the appropriate function to use if you want to GC everything and do *not* plan to resume work after the GC. That is, you should not have any work in progress when you execute **full-gc** (see the description of **gc-immediately**). If automatic GC is on when **full-gc** is called, it is turned off while **full-gc** executes and then turned on again when execution has completed.

Note that **full-gc** is intended for use before a **disk-save**. **full-gc** does some cleanup work to minimize the size of the disk-saved load band.

The **full-gc** function runs a before-full-GC initialization list to kill and free various processes, windows, buffers, resources, history lists, and so on that might have accumulated during this session. This operation makes the garbage-collected load band as small as possible by both getting rid of garbage and getting rid of nongarbage that you do not want after you reboot.

Anything killed or turned off by the **:full-gc** initialization list before a **disk-save** is typically recreated or turned back on by the **:warm** initialization list after reboot.

If **:before-disk-save** is true (the default), then **full-gc** makes certain preparations that should be made before a disk save (for example, dismounting the file system and clearing the namespaces), thereby collecting more nongarbage garbage. If **:duplicate-pnames** is true, then **full-gc** also collapses duplicate symbol print names so that the redundant strings can be collected. The default is **nil**.

The actual GC step is performed by default on *all* generations without promotion. After the GC, an after-full-GC initialization list is run. Note that a full GC may become impossible after a certain amount of garbage has accrued in the system because not enough free virtual memory is left to use for copying data.

:max-gen — The number of the oldest generation to be collected. The default is 3, which specifies to collect all generations.

:promote — Controls placement of the data that survives the GC operation in a given generation. If the value for this keyword is true, then data surviving GC in one generation is promoted to the next higher generation. Since promotion into a generation is done before that generation is collected, all nongarbage from all collected generations is placed in the generation above the value of **:max-gen** or in generation 3, whichever is smaller. If the value for **:promote** is false, surviving data is left where it was found. The default value for **full-gc** is **t**.

:silent — When the value of this keyword is true, no notifications are given during the execution of gc-immediately. The default is to notify as each generation is collected. The default is **nil**. A true value also suppresses user queries by GC. See **gc-immediately** for more information.

The **:max-gen** and **:promote** arguments can be used as with **gc-immediately**. However, it does not usually make sense to use **full-gc** to collect only the youngest generations. In general, you will want to invoke **full-gc** in one of the following ways:

- `(full-gc :duplicate-pnames t)`

  This does a complete garbage collection and takes quite a long time (at least 30 minutes). It requires that you have a large amount of free space to copy surviving generation 3 objects. You may not have enough free space to complete such a collection.

- `(full-gc :max-gen 2)`

  You can use this to collect only generations 0, 1, and 2, which will be much faster and require considerably less free address space. Specifying **full-gc** instead of **gc-immediately** will still release large data structures for collection. This will be a promoting collection.

Either of these is an appropriate sequence to use after you perform one or more **make-systems** and intend to save a band.

gc-and-disk-save *partition* &optional *unit* &key **:partition-comment** **:no-query** — Function

This function performs a complete garbage collection followed by a disk save. It is equivalent to the following:

```
(full-gc :duplicate-pnames t)
(disk-save partition unit :no-query t :partition-comment
partition-comment)
```

gc-and-disk-save warns you about the partition you are using and any space problems it detects. However, if the space problems are not solved by the garbage collection, the subsequent **disk-save** will not occur.

If **:no-query** is **t**, no questions are asked. This option should be used with caution.

sys:*gc-notifications* — Variable
sys:gc-report-stream — Variable

The **sys:*gc-notifications*** variable determines which automatic and batch GC actions are to attempt output messages to the user, as follows:

- **:batch-only** — Sends notifications for batch-style collections such as **full-gc** and **gc-immediately** but not for automatic collections. This is the default.
- **t** — Sends notifications for all automatic and batch collections.
- **nil** — Suppresses all notifications.

The **sys:gc-report-stream** controls where the GC messages are output, as follows:

- **t** — Messages are output to the current ***standard-output*** using **tv:notify**. This is the default.
- **nil** — Suppresses all messages.
- Any other value — The value is a stream to be used for output.

sys:*gc-daemon-notifications* — Variable
sys:gc-daemon-report-stream — Variable

If both of these variables are true, then the user is warned via a notification of low amounts of address space or low amounts of swap space. If either variable is false, no warnings are posted.

Automatic GC Functions and Variables

25.11.4 Regardless of whether automatic GC is on or off, almost all consing is done in generation 0. If automatic GC is off, everything remains in generation 0 until a user turns on automatic GC or calls for a batch GC.

If automatic GC is on, it monitors the size of the generations numbered less than or equal to the value of **sys:*gc-max-incremental-generation*** as they collect data. Whenever a generation reaches the threshold specified for that generation, it is garbage collected and its surviving data is promoted to the next higher generation (or to generation 2, whichever is smaller).

A given generation is normally garbage collected several times before the next higher generation reaches its threshold. By way if contrast, **gc-immediately** simply collects all generations from 0 to the value of its **:max-gen** argument, regardless of their sizes or thresholds.

gc-on Function
gc-off Function
sys:*gc-max-incremental-generation* Variable

These functions turn automatic GC on and off, respectively. Turning on automatic GC enables idle scavenging (see the description of **sys:inhibit-idle-scavenging-flag**) and adds a call to **gc-on** to the warm-boot initialization list. That is, if the load band is saved after a call to **gc-on**, then automatic GC is turned on by default in the new load band.

Turning automatic GC off forces any current collection to complete, disables idle scavenging, and deletes **gc-on** from the initialization list. That is, if a load band is saved after a call to **gc-off**, then automatic GC is turned off in the new load band. Depending on the amount of work necessary to complete the previous collection, **gc-off** may take a minute or more to return.

The value of the **sys:*gc-max-incremental-generation*** variable specifies the highest generation number that automatic GC will collect. The default is 2, and is limited to a maximum of 2. This variable provides **gc-on** with similar information as the **:max-gen** argument provides to **gc-immediately**, except that this variable can be changed in real time.

sys:*gc-console-delay-interval* Variable

If **sys:inhibit-scavenging-flag** and **sys:inhibit-idle-scavenging-flag** are both false (the system defaults), then the value of this variable specifies how soon scavenging starts after the console becomes idle, as follows:

- **:infinite** — Waits forever (effectively inhibits idle-time scavenging).
- **t** — Same as **:infinite**.
- **nil** — Starts scavenging immediately.
- Any integer — Specifies the number of seconds to wait before scavenging begins. The default is 30 seconds.

Load Band Training

25.11.5 A common but annoying characteristic of all virtual memory systems is that when you touch an object for the first time, there is a pause while it is paged into memory. Of course, when you bring in an object from disk, you actually bring in a whole page full of objects that presumably are also in your working set.

Unfortunately, the grouping of objects into pages on a load band reflects the system-build steps more strongly than it does your usage patterns. Even if you are using a small working set, the individual objects still must be paged in one per page from all over the load band. You have to work on a machine for several minutes before you can get it to work the way you prefer.

However, you can use GC as a working-set assembly tool rather than just as a way to reclaim memory. You can train your load band so that all objects of your working set are packed together in relatively few pages. You cannot avoid the basic paging delay, but now virtually all objects in that page are of interest to you rather than just one or two, as before. Using a trained band, you can get your machine to work as you prefer in just a few minutes.

The load band shipped with the Explorer system is pretrained for common activities such as the Lisp Listener, and the Zmacs editor. You should not need to retrain your band unless you are a vendor supplying major software application that runs in a very different environment than the Explorer system that is shipped. To train a load band, do the following:

1. Cold boot your system (log in, load any patches, and so on) and then load your application if it is not already present in the load band.

2. Issue the **sys:start-training-session** function.

3. Use your application (and the rest of the system) as you plan to use it during a normal working session.

4. Perform administrative clean-up by deleting anything in the system and in your application that was used during the training session but that you do not want saved in the trained load band (for example, Zmacs buffers, temporary windows, caches, and so on). This step is very important and varies considerably from environment to environment. The load band can become very large if structures created by your application remain in the band.

5. Issue the **sys:end-training-session** function. At this point, the equivalent of two **full-gc** steps are done. This takes quite a while, typically an hour or more.

6. Save your newly trained load band.

Any garbage you created during the training session is collected just as it always is. However, the generational bookkeeping now contains an explicit record of any nongarbage you consed up and paged in during the training session. This nongarbage is your working set.

If you are planning to create a trained band for delivery of your application, you may want to consult your Explorer technical support group for assistance.

sys:start-training-session Function
sys:end-training-session Function

All code and data objects referenced between calls to these two functions tends to occupy contiguous virtual memory addresses and becomes a working set on the load band. Since the first few paging actions after a cold boot of a trained band tend to bring in the entire working set—even the parts that have not been touched yet, your system will be much more responsive following a boot.

If automatic GC was on when the training session started, it is turned off during the training session and then turned back on when the session is over.

TGC Tuning

25.11.6 The following functions and variables can be used to monitor TGC activity and to tune it. Be aware, however, that TGC is manipulating the foundations of the Explorer environment and any mistakes made during tuning experiments may be irreversible. These symbols are considered part of the internal implementation of TGC and may change in future releases.

sys:gc-fraction-of-ram-for-generation-zero Variable

This variable is intended to establish a threshold such that there is always room for generation 0 in physical memory, whereas older generations are usually on disk. The value of this variable must be a floating-point number less than 1.0 and represents the threshold size for generation 0 expressed as a fraction of the installed physical memory. For example, if the value of this variable is 0.5, then the threshold size for generation 0 is one-half of the installed physical memory. The default is 0.1.

sys:inhibit-scavenging-flag Variable

If this variable is true, no scavenging of any kind is performed. If this variable is false, then during GC several words are scavenged each time new data is consed. The scavenging is proportional to the number of words consed (for example, one word is scavenged for every *n* words consed). Scavenging may also be allowed during console idle time (see the description of **sys:inhibit-idle-scavenging-flag**). This variable is manipulated by the **gc-on** and **gc-off** functions.

sys:inhibit-idle-scavenging-flag Variable

If this variable is true, GC scavenging is not allowed when the system is idle. If the value of this variable is false (the default) and if **sys:inhibit-scavenging-flag** is false, then scavenging is allowed after the console has been idle for the amount of time specified by **sys:*gc-console-delay-interval***.

sys:gc-idle-scavenge-quantum Variable

The value of this variable is the quantum parameter used by the GC process when scavenging during console idle time. This variable is one of the ways of adjusting the amount of scavenging allowed to intrude on foreground operation. The default is to scavenge 50 words at each idle invocation.

sys:inhibit-gc-flips &body *body* Macro
sys:arrest-gc Function
sys:unarrest-gc Function

The GC algorithms must necessarily assume an internally consistent set of pointers in the system. This required consistency is automatically guaranteed for everything *except* subprimitive functions that manipulate pointer directly. Therefore, all code employing subprimitives must either be executed within the **sys:inhibit-gc-flips** macro or be executed with the GC process arrested.

sys:%gc-generation-number Variable

The value of this variable is the number incremented on each flip. This variable should be treated as read-only because its value is crucial to the correct operation of the GC bookkeeping.

Resources

25.12 The traditional way of allocating and deallocating memory in Lisp is to create an object when it is needed, use the object, and then let the garbage collector reclaim it when it can no longer be accessed. Using this GC approach, the most direct way of managing memory would be to set an object to **nil** when it is no longer needed so that it becomes garbage immediately. However, GC-dependent memory management has two major drawbacks:

- If the object is complex and takes a long time to create, then having to create a new object every time one is needed can cause noticeable pauses in program execution.
- If automatic GC is not efficient, then users tend to turn it off; thus, garbage is never collected until the user calls for a batch GC.

GC efficiency has two aspects: how much automatic GC intrudes on the user's work, and the rate of automatic collection. Previous implementations of automatic GC degraded system operation such that many users never turned GC on, making GC rate irrelevant..

There is no acceptable memory management alternative for the automatic allocation of objects such as individual cons cells that happens as a side effect of system operation. However, there is an alternative for a temporary major data structure, such as a buffer or an array that the programmer explicitly creates under well-defined circumstances: the programmer can define a *resource* of that structure to be reused as needed.

The basic operation of a resource is simple. The programmer defines a constructor function and an initializer function for the resource. From then on, the program calls **allocate-resource** to access a preinitialized copy of the object to work with and then calls **deallocate-resource** when the object is no longer needed. When an object is deallocated, the resource places it on an available list rather than letting it become garbage. When **allocate-resource** is called, the resource returns a previously deallocated object, if one is available, or constructs a new one. In either case, the initializer function makes certain that the object returned by **allocate-resource** is suitable for use as a new object.

Use of a resource is usually preferred when the time needed to create a new object is significantly more than the time to reinitialize an existing object. Windows, especially frames, are objects that fall into this category. They take a considerable time to construct, but initializing them is often little more than clearing the screen.

On the other hand, every allocated object is initialized regardless of whether it was newly constructed or reused. Therefore, if the initialization step is lengthy, then use of resources does not save any time.

Resources are also preferred if the use of a given data structure is known to create a great amount of garbage per unit time. If the user has automatic GC turned off, then many data structures fall into this category.

Fortunately, the new Temporal Garbage Collection implementation of automatic GC on the Explorer system is efficient enough for most users to keep it running most of the time. Therefore, a given data structure should be placed in a resource only if its allocation/deallocation rate is high and sustained. Network buffers usually fall into this category. With TGC, the number of cases in which a resource is preferred for high-usage data structures is significantly lower than before.

Just as the advent of TGC has changed the trade-offs of when and when not to use resources, TGC has also changed the notion of *how* resources should be treated when they are used. Consider a resource that normally has only two or three allocated objects at a time. Suppose a momentary outburst of activity demanded the allocation of 20 objects at once. Later, activity drops down to normal levels. The 17 to 18 extra objects have become nongarbage garbage. That is, no one will need those extra objects for a long time (if ever); yet they are not collectible garbage because they are readily accessible on the resource.

A related situation occurs if a resource is used in an early portion of a program and then becomes idle. Thus, most of the time, that resource maintains all of its previously allocated objects on its deallocated list as nongarbage garbage. As an aid to resource housekeeping, the **reinitialize-resource** function causes all currently deallocated objects in a resource to become garbage. Resource objects that are still allocated are left untouched. Meanwhile, automatic GC finds the freed objects and makes their memory available for the rest of the program.

Even in a resource whose deallocated objects are too small to bother reclaiming as garbage, the temporarily deallocated objects can still contribute significantly to nongarbage garbage because of what those objects are holding. For example, consider a resource of small objects, each containing a pointer to other objects, such as processes, windows, screen arrays, and so on. Now suppose that each time one of these objects is allocated, the initialization function replaces the old processes, windows, and buffers with new ones.

In other words, each of these relatively insignificant resource objects is anchoring major data structures, thereby keeping those structures from becoming garbage. Yet all of these carefully conserved structures are immediately discarded as soon as the object with the pointer is reallocated. The programmer who defines a resource can help this situation by defining a deallocator function for that resource whose purpose is to kill large data structures carried in the object when these structures will not be needed by the next allocation. This deallocator function should be a complement of the initialization function: whatever is initialized at allocation time should be killed at deallocation time.

Defining Resources

25.12.1 The following functions define and allocate resources.

defresource *resource-name parameters* &optional *doc-string* Macro
&key **:constructor :free-list-size :initial-copies :initializer**
:deallocator :finder :matcher :checker

This macro defines a new resource. A data structure is created to record special information about the resource. This data structure is stored as the value of the **defresource** property on the symbol name. Several of the keywords to **defresource** define functions or forms that access this data structure. The **defresource** data structure is described in more detail in paragraph 25.12.2, Accessing the Resource Data Structure.

The *resource-name* argument must be a symbol. It is the name of the resource (such as **w:inspect-frame-resource**) and gets a **defresource** property of the internal data structure representing the resource.

The *parameters* argument is a lambda list giving names and default values of parameters to an object of this type. It is used in conjunction with the **:constructor, :checker, :matcher, :deallocator,** and **:initializer** keywords described later. For example, if you have a resource of two-dimensional arrays used for temporary storage in a program, that resource typically has two parameters: the number of rows and the number of columns. In the simplest case, *parameters* is **nil**.

The *doc-string* argument should be a string that is associated with the resource. It can be accessed (and updated using **setf**) by the **documentation** function with a *doc-type* of **'defresource**.

:constructor — This required keyword is responsible for making an object. It is used when you attempt to create an object from the resource when no suitable free objects exist.

The value of **:constructor** is either a form or the name of a function. If its value is a form, **:constructor** can access the *parameters* as variables. If its value is a function, it is given the internal data structure for the resource and any supplied *parameters* as its arguments. It must default any unsupplied optional parameters.

:free-list-size — This keyword determines the number of objects that the resource data structure initially has room to remember. However, this limit is not fixed because the data structure is enlarged if necessary.

:initial-copies — This keyword determines the number of objects that are made as part of the evaluation of **defresource.** As such, this keyword is useful to set up a pool of free objects during the loading of a program. The value of **:initial-copies** is a number (or **nil,** meaning 0). The default value is **nil.**

:initializer — The value of this keyword is a function that cleans up the contents of an object before each use and is called or evaluated each time an object is allocated, whether it is simply constructed or is being reused.

The value of **:initializer** is a form or a function (as with **:constructor**). In addition to the *parameters,* a form can access the variable **object** (in the current package). If a function is supplied, then the first argument is the resource data structure, followed be the object to initialize and the parameters.

:deallocator — The value of this keyword is a function that cleans up the contents of an object after each use. This function is called or evaluated each time an object is deallocated. The purpose of this form is to remove pointers to unused structures so that they can be garbage collected. Many applications may find it advantageous to initialize resource objects in both the **:constructor** and **:deallocator** forms instead of using the **:initializer** form.

The value of **:deallocator** is a form or a function (as with **:constructor**). In addition to the parameters, a form can access the variable **object** (in the current package). If a function is specified, the first argument is the resource being returned, the second argument is the resource overhead structure, and the remaining arguments are *parameters.*

:finder — The value of this keyword is a form or function that *finds* a resource. This option is useful whenever the resource being allocated is scarce in some sense; that is, whenever allocating the resource is not just a matter of allocating memory.

Specifically, when a resource must be allocated, the **:finder** function is called with scheduling disabled; the **:constructor** function is never called. This function receives the same arguments that the **:constructor** function would. The returned value of the **:finder** function should be the resource object.

When this option is used, the unallocated list of resource objects is not maintained by the system. If such maintenance is necessary, you will have to coordinate the **:finder** and **:deallocator** functions to maintain this list.

:matcher — The value of this keyword is a form or function that verifies that a particular resource object is acceptable according to the supplied *parameters*. If the pool of available resource objects is not sufficiently homogeneous according to the parameters with which they were created, it may be useful for you to define this tolerance function to determine if some existing unallocated resource is a match or is sufficient for the current request.

Specifically, when a resource request is made, the matcher function is called, with scheduling inhibited, to compare the creation parameters of an existing unallocated resource with the parameter for the pending request. The arguments of the **:matcher** function are the resource data structure, the candidate resource object, and the pending request parameters. If a form is supplied instead of a function, the variable **object** is bound to the candidate resource. The default matcher action is to compare the two parameter lists using **equal**.

:checker — The value of this keyword is a function that determines whether or not an object is safe to allocate. If no **:checker** is supplied, a default checker only checks to see if the resource is currently in use. Therefore, if an object has been allocated and has not been freed, it is not safe to use; otherwise, it is. The **:checker** keyword is called inside a **without-interrupts**.

The value of this keyword is a form or a function. In addition to the *parameters*, a form here can access the variables **object** and **in-use-p** (in the current package). A function receives these as its second and third arguments, after the data structure and before the *parameters*.

:cleanup — The value of this keyword is a function that is called before a **full-gc** to clean out the resources. The function is called with one argument, which is the resource name. If **:cleanup** is **nil**, no cleanup function is called.

The default **:cleanup** function is **sys:reinitialize-resource**, which removes all unallocated resource objects so they can be garbage-collected. If this causes a resource to have less than what the **defresource :initial-copies** options specified, then new resource objects are allocated. Unused objects are not kept in the resource because new copies are less likely to contain pointers to other structures which could be garbage-collected.

If these options are used with forms (as opposed to functions), the forms are compiled inline as part of the expansion of **defresource**, which is more efficient.

Most of the options are not used in typical cases, as in the following:

```
(defresource two-dimensional-array  (rows columns)
          :constructor (make-array  (list rows columns)))
```

Suppose that in this example, the array is to be 100 by 100, and you want to preallocate one array during the loading of the program so that you do not need to spend extra time creating an array when the need arises. You might simply use the following after your **defresource**, which would allocate a 100 by 100 array and then immediately free it:

```
(using-resource (foo two-dimensional-array 100 100))
```

Alternatively, you can write the following:

```
(defresource two-dimensional-array
    (&optional (rows 100) (columns 100))
       :constructor (make-array (list rows columns))
       :initial-copies 1)
```

Following is an example depicting how the **:matcher** option can be used. Suppose you want to have a resource of two-dimensional arrays (as in the previous example) except that when you allocate one dimension, you are not concerned with its exact size, only that it is large enough. Furthermore, you realize that you are going to have many different sizes. This poses the problem that if you always allocate an array of a specific size, you allocate a large number of arrays, seldom having a reusable one of the correct size. To counter this problem, you can use the following code:

```
(defresource sloppy-two-dimensional-array (rows columns)
    :constructor (make-array (list rows columns))
    :matcher (and (>= (array-dimension 1 object) rows)
                (>= (array-dimension 2 object) columns)
                                     ))
```

allocate-resource *resource-name* &rest *parameters* Function

This function allocates an object from the resource specified by *resource-name*. The various forms and/or functions given as options to **defresource**, together with any *parameters* given to **allocate-resource**, control how a suitable object is found and whether a new one must be constructed or an old one can be reused.

Note that the **using-resource** macro is normally preferable to **allocate-resource** itself. See the description of **using-resource** later in this paragraph.

deallocate-resource *resource-name resource* Function

This function frees the object *resource* and returns it to the free-object list of the resource specified by *resource-name*.

reinitialize-resource *resource-name* Function

This function calls **clear-resource** on each unused resource in *resource-name*. It then allocates as many new resources as necessary to comply with the **:initial-copies** argument of **defresource**.

The primary purpose of this function is to reduce the amount of virtual memory currently allocated to this resource. For instance, if there was a peak demand for this resource that has subsequently disappeared, it may be practical to get rid of some of the unallocated resources. Also, unallocated resources can contain pointers to structures that otherwise would be garbage collected. Because of this, even unallocated resources are cleared and then reallocated.

clear-resource *resource-name* &optional *instance warning-p* Function

This function eliminates all pointers to those objects remembered by the resource specified by *resource-name*. Future calls to **allocate-resource** create new objects. This function is useful if something about the resource has been changed incompatibly so that the old objects are no longer usable. If *instance* is supplied, it should be an instance of *resource-name*, in which case only that instance is cleared. If *warn-p* is true (the default), then a warning message is printed to the value of ***error-output*** if an instance being cleared is currently allocated.

using-resource *(variable resource-name parameters...)* &body *body...* Macro

This macro sequentially evaluates the *body* forms with *variable* bound to an object allocated from *resource-name*, using the given *parameters*. The *parameters*, if any, are evaluated, but *resource* is not.

The **using-resource** macro is often more convenient to use than the **allocate-resource** and **deallocate-resource** functions. Furthermore, it is careful to free the object when the body is exited, whether it returns normally or via a **throw**. This operation is performed by using **unwind-protect**. For information about **unwind-protect**, see Section 14, Control Structures. The following example depicts one use of **using-resource**:

```
(defresource huge-16b-array (&optional (size 1000))
  :constructor (make-array size :element-type '(unsigned-byte 16))

(defun do-complex-computation (x y)
  (using-resource (temp-array huge-16b-array)
    ...                    ;Within the body, the array can be used.

    (setf (aref temp-array i) some-number)
    ...))     ;The array is returned at the end.
```

deallocate-whole-resource *resource-name* Function

This function frees all objects in *resource-name*. This function is like executing **deallocate-resource** on each object individually. The **deallocate-whole-resource** function is often useful in warm-boot initializations.

map-resource *function resource-name* &rest *extra-args* Function

This function calls *function* on each object created in *resource-name*. Each time *function* is called, it receives three fixed arguments plus whatever *extra-args* were specified. The three fixed arguments are an object of the resource, **t** if the object is currently allocated (that is, in use), and the resource data structure itself.

Accessing the Resource Data Structure

25.12.2 The **constructor, initializer, matcher, deallocator** and checker functions receive the internal resource data structure as an argument. This is a named structure array whose elements record the objects, both free and allocated, and whose array leader contains other information. To access this structure, use one of the following primitives.

sys:get-resource-structure *resource-name* Function

This function returns the resource structure for the resource named *resource-name*.

sys:resource-object *resource-structure index* Function

Of the objects remembered by the resource, this function returns the object specified by *index*. Both free and allocated objects are remembered.

sys:resource-in-use-p *resource-structure index* Function

From the objects remembered by the resource, this function returns true if the object specified by *index* has been allocated and not deallocated. Simply defined resources do not reallocate an object in this state.

sys:resource-parameters *resource-structure index* Function

This function returns the list of parameters from which the object specified by *index* was originally created.

sys:resource-n-objects *resource-structure* Function

This function returns the number of objects currently remembered by the resource. This includes all objects ever constructed, unless **clear-resource** has been used.

STACK GROUPS

26

Stack Group Definitions

26.1 The data type **stack-group** (usually abbreviated SG) is a type of Lisp object useful for implementing certain advanced control structures such as coroutines and generators. Processes, which are a kind of coroutine, are built on top of stack groups (see Section 27, Processes). A stack group represents a computation and its internal state, including the Lisp stack.

At any time, the computation being performed by the Explorer system is associated with one stack group, called the *current* or *running* stack group. The operation of designating a particular stack group to be the current stack group is called a *resumption* or a *stack group switch*. The previously running stack group is said to have *resumed* the new stack group. The *resume* operation has two parts: first, the state of the running computation is *saved away inside* the current stack group; second, the state saved in the new stack group is restored, and the new stack group is made current. Then the computation of the new stack group resumes its course.

The stack group itself holds a great deal of state information. It contains the control stack, or *regular push down list* (*PDL*). The control stack is what you are shown in the debugger. The control stack remembers the function that is running, its caller, its caller's caller, and so on, and the point of execution of each function (the *return addresses* of each function). A stack group also contains the environment stack, or *special PDL*. Switching to a stack group moves the current bindings from the special PDL to the symbol value cells, exchanging them with the global or other shadowed bindings. Switching out of a stack group reverses the process. (The name *stack group* derives from the existence of these two stacks.) Finally, the stack group contains various internal state information (contents of machine registers and so on).

When the state of the current stack group is saved, all of its bindings are undone, and when the state is restored, the bindings are reinstated. Note that although bindings are temporarily undone, unwind-protect handlers are *not* run by a stack group switch (see the function **let-globally** in Section 2, Symbols).

Each stack group is a separate environment for purposes of function calling, throwing, dynamic variable binding, and condition signaling. All stack groups run in the same address space; thus, they share the same Lisp data and the same global (not lambda-bound) variables.

When a new stack group is created, it is empty; it does not contain the state of any computation, so it cannot be resumed. Before processing can begin, the stack group must be set to an initial state. This is achieved by *presetting* the stack group. To preset a stack group, you supply a function and a set of arguments. The stack group is placed in a state such that when it is first resumed, this function is called with those arguments. The function is called the *initial function* of the stack group.

Resuming of Stack Groups

26.2 The most noteworthy aspect of stack groups is that they resume each other. When one stack group resumes a second stack group, the current state of Lisp execution is saved in the first stack group and is restored from the second stack group. Resuming is also called *switching stack groups*.

At any time, the current computation is associated with one stack group, which is called the *current stack group*. The computations associated with other stack groups have their states saved in memory and are not computing. Thus, the only stack group that can do anything at all, particularly resuming other stack groups, is the current one.

Suppose computation A is executing and it resumes a stack group B. Computation A's state is saved into the current stack group, and computation B begins execution. Computation A now lies dormant in the original stack group, while computation B resumes computation F, which resumes computation D, which resumes computation E, and so on. At some time, some computation resumes the original stack group, and computation A is restored from the stack group to begin execution once again. The current stack group can resume other stack groups in several ways. This section describes all the various kinds of resumptions.

Associated with each stack group is a *resumer*. The resumer is **nil** or another stack group. Some forms of resuming examine and alter the resumer of some stack groups. The process of resuming can also transmit a Lisp object from the old stack group to the new stack group. Each stack group specifies a value to transmit whenever it resumes another stack group; whenever a stack group is resumed, it receives a value.

Voluntary Resumption

In the following descriptions, the term *current* stands for the current stack group, *another-sg* stands for some other stack group, and *object* stands for any arbitrary Lisp object.

Stack groups can be used as functions. They accept one argument. If *current* calls *another-sg* as a function with one argument *object*, then *another-sg* is resumed, and the object transmitted is *object*. When *current* is resumed (usually, but not necessarily, by *another-sg*), the object transmitted by that resumption is returned as the value of the call to *another-sg*. Calling a stack group as a function is one of the simple ways to resume the stack group. The value you transmit is the argument to the function, and the value you receive is the value returned from the function. Furthermore, this form of resuming sets *another-sg*'s resumer to be *current*.

Another way to resume a stack group is to use **stack-group-return**. Rather than allowing you to specify which stack group to resume, this function always resumes the resumer of the current stack group. Using this function is a good way to resume whichever stack group resumed your stack group, assuming that it was done by function calling. The **stack-group-return** function takes one argument, which is the object to transmit. It returns when another stack group resumes the current stack group. The value returned by **stack-group-return** is the object that was transmitted by that resumption. The **stack-group-return** function does not affect the resumer of any stack group.

The most fundamental way to resume is with **stack-group-resume**, which takes two arguments: the stack group and a value to transmit. It returns when the current stack group is resumed, returning the value that was transmitted by that resumption, and does not affect any stack group's resumer.

If the initial function of *current* attempts to return a value *object*, then the regular kind of Lisp function return cannot take place because the function did not have any caller (it was started when the stack group was initialized). Thus, instead of normal function returning, a stack group return happens. The resumer of *current* is resumed, and the value transmitted is *object*. Consequently, *current* is left in a state from which it cannot be resumed again (the exhausted state). Any attempt to resume it signals an error. Presetting it makes it work again.

Involuntary Resumption

The preceding are the *voluntary* forms of stack group switch; a resumption happens because the computation indicated that it should. There are also two *involuntary* forms in which another stack group is resumed without the explicit request of the running program.

If an error occurs, the current stack group resumes the error handler stack group. The value transmitted is partially descriptive of the error. The error handler looks inside the saved state of the erring stack group to obtain the rest of the information. The error handler recovers from the error by changing the saved state of the erring stack group and then resuming it.

When certain events occur, typically a 1-second clock tick, a *sequence break* occurs. This sequence break forces the current stack group to resume a special stack group called the *scheduler* (see Section 27, Processes). The scheduler implements processes by sequentially resuming the stack group of each process that is ready to run.

An Example Using Stack Groups

26.3 The canonical coroutine example is the so-called same fringe problem: given two trees, determine whether they contain the same atoms in the same order, ignoring parenthesis structure. In other words, given two binary trees built from conses, determine whether the sequences of atoms on the fringes of the trees are the same, ignoring differences in the arrangement of the internal skeletons of the two trees. Following the usual rule for trees, **nil** in the cdr of a cons is to be ignored.

One way of solving this problem is to use *generator* coroutines. First, make a generator for each tree. Each time the generator is called, it returns the next element of the fringe of its tree. After the generator has examined the entire tree, it returns a special exhausted flag. The generator is most naturally written as a recursive function. The use of coroutines, that is, stack groups, allows the two generators to recurse separately on two different control stacks without needing to coordinate with each other.

The program is very simple. To construct it in the usual bottom-up style, first write a recursive function that takes a tree and performs **stack-group-returns** for each element of its fringe. The **stack-group-return** is how the generator coroutine delivers its output. You can easily test this function by changing **stack-group-return** to **print** and trying it on some examples:

```
(defun fringe (tree)
  (if (atom tree)
      (stack-group-return tree)     ; Return this leaf node.
      (fringe (car tree))           ; Process leaf nodes on CAR branch.
      (unless (null (cdr tree))     ; Is the CDR branch empty?
        (fringe (cdr tree)))))      ; Process leaf nodes on CDR branch.
```

Now package this function inside another, which takes care of returning the special exhausted flag:

```
(defun fringe1 (tree exhausted)
 (fringe tree)                  ; root for recursive calls
 exhausted)                     ; after all recursion is finished
```

The `samefringe` function takes the two trees as arguments and returns **t** or **nil**. It creates two stack groups to act as the two generator coroutines, presets them to run the `fringe1` function, and then goes into a loop comparing the two fringes. The value is **nil** if a difference is discovered or **t** if they are still the same when the end is reached:

```
(defun samefringe (tree1 tree2)
    (let ((sg1 (make-stack-group "samefringe1"))
          (sg2 (make-stack-group "samefringe2"))

           (exhausted (list nil)))

       (stack-group-preset sg1 #'fringe1 tree1 exhausted)
       (stack-group-preset sg2 #'fringe1 tree2 exhausted)

       (loop
          for v1 = (funcall sg1 nil)
          for v2 = (funcall sg2 nil)
          do
            (when (not (eq v1 v2)) (return nil))
            (when (eq v1 exhausted) (return t))
        )
    )
)
```

Now test the **samefringe** function on two examples:

```
(samefringe '(a b c) '(a (b c))) => t
(samefringe '(a b c) '(a b c d)) => nil
```

A problem arises since a stack group is quite a large object, and you make two of them every time you compare two fringes. This process requires considerable overhead. It can easily be eliminated with a modest amount of explicit storage allocation, using the resource facility (see Section 25, Storage Management).

You can also avoid making the exhausted flag fresh each time; its only important property is that it should not be an atom:

```
(defresource samefringe-coroutine ()
  :constructor (make-stack-group "for-samefringe"))

(defvar exhausted-flag (list nil))

(defun samefringe (tree1 tree2)
 (using-resource (sg1 samefringe-coroutine)
   (using-resource (sg2 samefringe-coroutine)
     (stack-group-preset sg1 #'fringe1 tree1 exhausted-flag)
     (stack-group-preset sg2 #'fringe1 tree2 exhausted-flag)
     (loop
        for v1 = (funcall sg1 nil)
        for v2 = (funcall sg2 nil)
        do
          (when (not (eq v1 v2)) (return nil))
          (when (eq v1 exhausted) (return t))))))
```

Now you can perform as many comparisons as you want with the same amount of memory as would have been used for only one comparison.

Stack Group States

26.4 Each stack group has a *state*, which controls what the stack group does when it is resumed. The code number for the state is returned by the function **sys:sg-current-state**. This number is the value of one of the following constants. Only the states actually used by the current system are documented here; some other codes are defined but not used.

sys:sg-state-active Constant

The stack group is the current one.

sys:sg-state-resumable Constant

The stack group is waiting to be resumed, at which time it picks up its saved machine state and continues doing what it was doing before.

sys:sg-state-awaiting-return Constant

The stack group called another stack group as a function. When it is resumed, it returns from that function call.

sys:sg-state-invoke-call-on-return Constant

When the stack group is resumed, it calls a predetermined function. The function and arguments are already set up on the stack. The debugger uses this state to force the stack group being debugged to perform various actions.

sys:sg-state-awaiting-error-recovery Constant

When a stack group encounters an error, it goes into this state, which prevents anything from happening to it until the error handler has examined it. Meanwhile, it cannot be resumed.

sys:sg-state-awaiting-initial-call Constant

The stack group has been preset (see the **stack-group-preset** function later in this section), but it has never been called. When it is resumed, it calls its initial function with the preset arguments.

sys:sg-state-exhausted Constant

The stack group's initial function has returned and cannot be resumed until the stack group is once again preset. See **stack-group-preset** for details.

Stack Group Functions

26.5 The following functions are associated with stack groups.

make-stack-group *name* &key **:sg-area :regular-pdl-area :special-pdl-area :regular-pdl-size :special-pdl-size :swap-sv-on-call-out :swap-sv-of-sg-that-calls-me :trap-enable :safe** Function

This function creates and returns a new stack group. The *name* argument, which can be any symbol or string, is used in the stack group's printed representation. The options are not very useful because most calls to **make-stack-group** do not need any options at all.

:sg-area — The value of this keyword is an area in which the stack group structure is created. This option defaults to the area identified by the variable **default-cons-area**.

:regular-pdl-area — The value of this keyword is an area in which to create the regular PDL. Note that only certain areas are appropriate for this keyword because regular PDLs are cached in a hardware device called the *PDL buffer*. The default is **sys:pdl-area**.

:special-pdl-area — The value of this keyword is an area in which to create the special PDL. This option defaults to the area identified by the variable **default-cons-area**.

:regular-pdl-size — The value of this keyword is an integer that determines the length of the regular PDL to be created. This keyword defaults to 1536.

:special-pdl-size — The value of this keyword is an integer that determines the length of the special PDL to be created. This keyword defaults to 1024.

:swap-sv-on-call-out, :swap-sv-of-sg-that-calls-me — These keywords default to 1. If these are 0, the system does not maintain separate binding environments for each stack group. Do not use this feature.

:trap-enable — The value of this keyword determines what to do if a microcode error occurs. If it is 1, the system tries to handle the error; if it is 0, the machine halts. The keyword defaults to 1. It is 0 only in the error handler stack group, a trap in which it would not work anyway.

:safe — If this flag is 1 (the default), a strict call-return discipline among stack groups is enforced. If it is 0, no restriction on stack group switching is imposed.

stack-group-preset *stack-group function* &rest *arguments* Function

This function sets up *stack-group* so that when it is resumed, *function* is applied to *arguments* within the stack group. Both stacks, the regular and special PDLs, are reset and any previously saved information is lost. The **stack-group-preset** function is typically used to initialize a stack group immediately after it is made, but it can be used on any stack group at any time. If you execute this function on a stack group that is not exhausted, its present state is destroyed without properly cleaning up by running the clean-up forms of **unwind-protects**.

stack-group-resume *stack object* Function

This function resumes *stack*, transmitting the value *object*. No stack group's resumer is affected.

sys:sg-resumable-p *stack* Function

This function returns true if the state of *stack* permits it to be resumed.

sys:wrong-stack-group-state (error) Condition

This condition is signaled if, for example, you try to resume a stack group that is in the exhausted state. The **sys:wrong-stack-group-state** condition is based on the **error** flavor.

stack-group-return *object* Function

This function resumes the current stack group's resumer, transmitting the value *object*. No stack group's resumer is affected.

symeval-in-stack-group *symbol sg* &optional *frame as-if-current-p* Function

This function evaluates the *symbol* in the binding environment of *frame* in stack group *sg*. A frame is an index in the stack group's regular PDL. If *frame* is **nil**, the current frame in *sg* is used. If *frame* is 0, the environment is global. The *frame* argument defaults to the stack group's current frame.

This function returns three values:

- The value of *symbol*.
- A non-**nil** value (actually a copy of the third value) that indicates that *symbol* was bound.
- The locative indicating where the value is stored, or **nil** if there is no location. If the value of *as-if-current-p* is true, the locative points to where the value would be stored if *sg* was running. This location can be different from where the value is currently stored. For example, the current binding in stack group *sg* is stored in the value cell of *symbol* when *sg* may be running, but the value of it is stored in the special PDL of *sg* when *sg* is not running. If *as-if-current-p* is **nil** (the default), the first value is the current value, not what is currently stored in the locative. When *sg* is the current stack group, *as-if-current-p* is ignored.

If *symbol* is unbound in *sg* and *frame*, the first and second values are both **nil**. The third value is still the locative.

NOTE: Do not call this function if *sg* might be running in another process and might be changing its state.

---

current-stack-group-resumer Variable

This variable is bound to the resumer of the current stack group.

current-stack-group Variable

The value of this variable is the stack group that is currently running. A program can use this variable to access its own stack group.

sys:pdl-overflow (error) Condition

This condition is signaled when there is overflow on either the regular PDL or the special PDL. The **:pdl-name** operation on the condition instance returns either **:special** or **:regular**, which tell handlers where the overflow occurred.

The **:grow-pdl** proceed type is provided. It takes no arguments. Proceeding from the error automatically makes the affected PDL bigger. The **sys:pdl-overflow** condition is based on the **error** flavor.

eh:pdl-grow-ratio Variable

This variable is the factor by which to increase the size of a PDL after an overflow. It is initially 1.5.

Analyzing Stack Frames

26.6 A stack frame is represented by an index in the regular PDL array of the stack group. The stack frame contains information about the current function, the arguments it was called with, and the local variables used within the function.

sys:sg-regular-pdl *sg* Function

This function returns the regular PDL of *sg*. This function is an array of type **art-reg-pdl**. Stack frames are represented as indices into this array.

sys:sg-regular-pdl-pointer *sg* Function

This function returns the index in the *sg* argument's regular PDL of the last word pushed.

sys:sg-special-pdl *sg* Function

This function returns the special PDL of *sg*. This function is an array of type **art-special-pdl**, used to hold special bindings made by functions executing in that stack group.

sys:sg-special-pdl-pointer *sg* Function

This function returns the index in the *sg* argument's special PDL of the last word pushed.

Internal Stack Frame Functions

26.7 The following functions are used to move from one stack frame to another when a stack group's regular PDL is examined.

eh:sg-innermost-frame *sg* Function

This function returns the regular PDL frame of the innermost frame in *sg*, the one that would be executing if *sg* were current. If *sg* is current, the value is the frame of the caller of this function.

eh:sg-next-frame *sg frame* Function

This function returns the next frame out from *frame* in *sg*. The next frame is the one that called *frame*. If *frame* is the outermost frame, the value is **nil**.

eh:sg-previous-frame *sg frame* &optional *innermost* Function

This function returns the previous frame in from *frame* in *sg*. The previous frame is the one called by *frame*. If *frame* is the currently executing frame (the innermost frame), the value returned is **nil**. If *frame* is **nil**, the value returned is the outermost or initial frame. If *innermost* is specified and is a frame in *sg*, it is used as the innermost frame.

eh:sg-previous-nth-frame *sg frame* &optional *n innermost* Function

This function moves up or down *n* frames in the stack group *sg*, starting at *frame*. If *n* is positive (it defaults to 1), the function moves inward and returns the *n*th next frame. If *n* is negative, the function returns the *n*th previous frame. If *innermost* is specified and is a frame in *sg*, it is used as the innermost frame; all frames inside the innermost are ignored. If the function reaches the top or the bottom of the stack, then it returns the innermost or outermost frame accordingly. If the specified number of frames is passed, the second value is **nil**.

Running interpreted code involves calls to **eval**, **cond**, and so on, which are not present in compiled code. The following four functions can be used to skip over the stack frames of such functions, showing only the frames for the functions the user knows.

eh:sg-previous-nth-interesting-frame *sg frame* &optional *n innermost* Function

This function is similar to **eh:sg-previous-nth-frame**, but it skips over uninteresting frames. It ignores those frames used by the interpreter and special forms such as **prog** and **let**.

eh:sg-previous-interesting-frame *sg frame* &optional *innermost* Function

This function is similar to **eh:sg-previous-frame**, but it skips over uninteresting frames. It ignores those frames used by the interpreter and special forms such as **prog** and **let**.

eh:sg-next-interesting-frame *sg frame* Function

This function is similar to **eh:sg-next-frame**. It returns the next *frame* out of the stack group *sg*, but it skips over uninteresting frames. It ignores those used by the interpreter and special forms such as **prog** and **let**.

eh:sg-out-to-interesting-frame *sg frame* Function

If *frame* in the stack group *sg* is not interesting, this function finds a frame outside of it that is interesting. It either returns the value *frame* or the index of a frame outside of *frame*.

The following functions are used to analyze the data in a particular stack frame.

sys:rp-function-word *regpdl frame* Function

This function returns the function executing in *frame*. The *regpdl* argument should be the **sg-regular-pdl** of the stack group.

eh:sg-number-of-spread-args *sg frame* Function

This function returns the number of arguments received by the active *frame* in the stack group *sg*. The **-spread-args** part of the function name means that the rest argument and arguments received by it are not included. If this function is called with keywords, the keyword symbol and the associated value are counted as separate arguments. For example:

```
(setq sg (make-stack-group 'sg))     ; Make a stack group.
(defun test (a b c &key e f) nil)    ; Define a dummy function.
(stack-group-preset sg 'test 1 2 3 :e 5 :f 6)
(eh:sg-number-of-spread-args sg (eh:sg-innermost-frame sg)) => 7
```

In this example, the innermost frame corresponds to the initial function because the stack group is not yet resumed.

eh:sg-frame-arg-value *sg frame n* &optional *errorp* Function

This function returns the value and the location of argument number *n* of the *frame* stack frame in the stack group *sg*. If *errorp* is true, an error is signaled if *n* is out of range.

The first value is the value of the specified argument. The second value is the location in which the argument is stored when *sg* is running. The location actually may not be in the stack if the argument is special. The location may then have other contents when the stack group is not running. If *errorp* is **nil** and an error occurred, the function returns a third value—a string describing the error. The *errorp* argument defaults to **t**.

eh:sg-rest-arg-value *sg frame* Function

This function returns the value of the rest argument in *frame*, or **nil** if there is none.

The second value is true if the function called in *frame* expects an explicitly passed rest argument.

The third value is true if the rest argument was passed explicitly. If this value is **nil**, the rest argument is a stack list that overlaps the arguments of the *frame* stack frame. (If passed explicitly, it may still be a stack list but not in this frame.)

eh:sg-number-of-locals *sg frame* Function

This function returns the number of local variables in the *frame* stack frame.

eh:sg-frame-local-value *sg frame n* &optional *errorp* Function

This function returns the value and the location of local variable number *n* of the *frame* stack frame in the stack group *sg*. If *errorp* is true, an error is signaled if *n* is out of range.

The first value is the value of the specified local variable. The second value is the location in which the local variable is stored when *sg* is running. The location actually may not be in the stack; if not, it may have other contents when the stack group is not running.

If *errorp* is **nil** and an error has occurred, the function returns a third value—a string describing the error.

eh:sg-frame-value-value *sg frame n* &optional *errorp* Function

This function returns the value and location of the *n*th multiple value that *frame* has returned. If *errorp* is true, an error is signaled if *n* is out of range.

If *errorp* is **nil** and an error has occurred, the function returns a third value—a string describing the error.

eh:sg-frame-value-list *sg frame* Function

This function returns two values that describe whether the *frame*'s caller wants multiple values as well as any values that *frame* is returning. The first value returned is a list containing the values being returned by *frame*. The second value can be one of the following three values:

- **nil** — Indicates that this **frame** was not invoked to return multiple values.
- A number — The number of values returned.
- A locative — Indicates that this frame was called with **multiple-value-list**.

The third of the preceding values is included for historical reasons; the value returned is always the beginning of the list being returned after all the values have been calculated.

eh:sg-frame-special-pdl-range *sg frame* Function

This function returns two values delimiting the range of *sg*'s special PDL that belongs to the specified stack frame. The first value is the index of the first special PDL word that belongs to the frame, and the second value is the index of the next word that does not belong to it.

If the specified frame has no special bindings, both values are **nil**. Otherwise, the indicated special PDL words describe bindings made on entry to or during execution in this frame. The words come in pairs.

The first word of each pair contains the saved value; the second points to the location that was bound. When the stack group is not current, the saved value is the value for the binding made in this frame. When the stack group is current, the saved value is the shadowed value, and the value for this binding is either in the cell that was bound or is the saved value of another binding, at a higher index, of the same cell.

The **sys:%%specpdl-closure-binding** bit is nonzero in the first word of the pair if the binding was made before entry to the function itself; this includes bindings made by closures and by instances (including **self**). Otherwise, the binding was made by the function itself (including the arguments that are declared special).

The **symeval-in-stack-group** function can be used to find the value of a special variable at a certain stack frame.

Input/Output in Stack Groups

26.8 Because each stack group has its own set of dynamic bindings, a stack group does not inherit its creator's value of ***terminal-io*** nor its caller's unless you make special provision. (For information about the variable ***terminal-io***, see the section entitled Streams in the *Explorer Input/Output Reference* manual.) The ***terminal-io*** that a stack group receives by default is a *background* stream that does not normally expect to be used. If it is used, it turns into a *background window* that requests the user's attention (usually resulting from an error printout that tries to print on the stream). For related information, see the *Explorer Window System Reference* manual.

If you write a program that uses multiple stack groups and you want them all to perform input and output to the terminal, pass the value of ***terminal-io*** to the top-level function of each stack group as part of the **stack-group-preset**. That function should bind the ***terminal-io*** variable.

Another technique to achieve these same results is to use a closure as the top-level function of a stack group. This closure can bind ***terminal-io*** and any other variables that should be shared between the stack group and its creator.

PROCESSES

27

Introduction

27.1 A process is a Lisp object to which the scheduler allocates CPU time. A process consists of at least one stack group containing the current state of the computation and additional overhead that is used by the scheduler to determine when and how a process runs. In general, stack groups represent the computation, and the process provides run-time context for that computation.

A typical standalone application needs to create a process, preset the initial stack group function, and provide a run reason for the process. The initial function creates any requirements, such as windows, that are needed by the application. The application can create additional stack groups that call one another sequentially. In a scenario like this, the running stack group is remembered by the current process. The next time the scheduler selects this process, the stack group computation is resumed. The application may also produce processes to run additional computations in parallel.

If all the processes are simply trying to compute, the machine time-slices between them. This is not an efficient mode of operation because dividing the finite memory and processor power of the machine among several processes cannot increase the available power and, in fact, wastes some of it in overhead. Generally speaking, there can be several ongoing computations, but at a given moment only one or two processes are trying to run. The rest are either *waiting* for an event to occur or are *stopped*; that is, they are not allowed to compete for resources.

A process waits for an event by means of the **process-wait** primitive, which is given a predicate function that detects the awaited event. A module of the system called the process scheduler periodically calls this function. If it returns **nil**, the process continues to wait; if it returns true, the process becomes runnable, and its call to **process-wait** returns, allowing the computation to proceed.

A process can be *active* or *stopped*. Stopped processes are never allowed to run; they are not considered by the scheduler and so never become the current process until they are made active again. The scheduler continually tests the waiting functions of all the active processes, and those that return non-**nil** values are allowed to run. When you first create a process with **make-process**, it is inactive.

A process has two sets of Lisp objects associated with it called its *run reasons* and its *arrest reasons*. These sets are implemented as lists. Any kind of object can be in these sets; typically, keyword symbols and active objects such as windows and other processes are found. A process is considered *active* when it has at least one run reason and no arrest reasons; that is, the run-reason list is non-**nil** and the arrest-reason list is **nil**.

To activate a computation in another process, you must first create a process, then specify which computation you want to happen in that process. The computation to be executed by a process is specified as an *initial function* and a list of arguments to that function. When the process starts up, it applies the function to the arguments. In some cases, the initial function is written so that it never returns, while in other cases it performs a certain computation and then returns, which stops the process.

To *reset* a process means to throw out its entire computation (see **throw** in Section 14, Control Structures), then force it to call its initial function again. Resetting a process clears its waiting condition, and thus if it is active, it is enabled to run. To *preset* a function is to set up its initial function (and arguments) and then reset it. This is how you start up a computation in a process.

All processes in the Explorer system run in the same virtual address space, sharing the same set of Lisp objects. Unlike other systems that have special restricted mechanisms for interprocess communication, the Explorer system allows processes to communicate in arbitrary ways through shared Lisp objects. One process can inform another of an event simply by changing the value of a global variable. Buffers containing messages from one process to another can be implemented as lists or arrays. The usual mechanisms of atomic operations, critical sections, and interlocks are provided (see **store-conditional**, **without-interrupts**, and **process-lock** later in this section).

A process is a Lisp object, an instance of one of several flavors of process (for information, see Section 19, Flavors). The remainder of this section describes the scheduler, the operations defined on processes, the functions you can apply to a process, and the functions and variables a program running in a process can use to manipulate its process.

Creating a Process

27.2 There are many parameters that can be used to configure a new process. In many cases, it is useful to accept the defaults for all arguments except the name and initial function. The **process-run-function** and **process-run-restartable-function** functions can be used in these cases. These functions allocate a process from a pool of unused processes. When the computation completes, the exhausted process returns to the pool. While these functions return the process object, do not maintain a pointer to these objects because they may be reallocated to another computation without notifying the process creator.

If you do not want the process defaults used in these functions or if you want to maintain a pointer to the process so that you can send messages to it, you must call **make-process** directly.

make-process *name* &key **:simple-p :flavor :stack-group :warm-boot-action :quantum :priority :run-reasons :arrest-reasons :sg-area :regular-pdl-area :special-pdl-area :regular-pdl-size :special-pdl-size :swap-sv-on-call-out :swap-sv-of-sg-that-calls-me :trap-enable** Function

This function creates and returns a process specified by *name*. The process is not capable of running until the initial function is set using the **:preset** operation and it is given a run reason using the **:run-reason** operation.

Keywords (which are optional) allow you to specify items about the process. The following keywords are allowed:

:simple-p — Specifying true for this keyword gives you a simple process (described later in this section).

:flavor — Specifies the flavor of process to be created. See paragraph 27.3, Process Flavors, for a list of all the flavors of process supplied by the system.

:stack-group — Identifies the stack group to be used by the process. If this option is not specified, a stack group is created according to the relevant options that follow.

:warm-boot-action — Specifies what to do with the process when the machine is booted (see paragraph 27.4, Process Generic Operations).

:quantum — See paragraph 27.3, Process Flavors.

:priority — See paragraph 27.3, Process Flavors.

:run-reasons — Allows you to supply an initial run reason. The default is **nil**.

:arrest-reasons — Allows you to supply an initial arrest reason. The default is **nil**.

:sg-area — This is the area in which to create the stack group. The default is the value of **default-cons-area**.

:regular-pdl-area
:special-pdl-area
:regular-pdl-size
:special-pdl-size — These options are passed on to **make-stack-group** (see paragraph 26.3, An Example Using Stack Groups).

:swap-sv-on-call-out and **:swap-sv-of-sg-that-calls-me** — These flags default to 1. If they are 0, the system does not maintain separate binding environments for each stack group. Do not use these options.

:trap-enable — Determines what to do if a microcode error occurs. If it is 1, the system tries to handle the error; if it is 0, the machine halts. The default value is 1. Its value is 0 only in the error handler stack group, a trap that would not work anyway.

The **make-process** function is defined with the &allow-other-keywords lambda-list keyword. When the process is created, all of the arguments for the keywords that are passed to **make-process** are sent as an *init-plist* to **instantiate-flavor**. If you defined your own process flavor, you can supply initialization keywords to **make-process** because they are passed along.

The following two functions allow you to call a function and have it execute asynchronously in another process. You can use one of these functions as a simple way to start up a process that runs *forever* or as a way to make something happen without the necessity of waiting for it to complete. When the function returns, the process is returned to a pool of free processes for reuse. The only difference between these three functions is in what happens if the machine is booted while the process is still active.

Normally, the function to be run should not perform any I/O to the terminal. For more information, see paragraph 26.8, Input/Output in Stack Frames.

process-run-function *name-or-options function* &rest *args* Function

This function creates a process, presets it so that it applies *function* to *args*, and starts it running.

The *name-or-options* argument can be either a string specifying a name for the process or a list of keywords that can specify the name and various other parameters:

:name — This keyword should be followed by a string that specifies the name of the process. The default is `"Anonymous"`.

:restart-after-reset — This keyword indicates what to do to the process if it is reset. Specifying **nil** means the process should be killed; anything else means the process should be restarted. The default is **nil**.

:warm-boot-action — This keyword indicates what to do with the process when the machine is booted. For more details, see paragraph 27.4, Process Generic Operations.

:restart-after-boot — This keyword provides a simpler way of indicating what to do with the process when the machine is booted. If the **:warm-boot-action** keyword is not supplied or its value is **nil**, then this keyword's value is used instead. Specifying **nil** means the process should be killed; anything else means the process should be restarted. The default is **nil**.

:quantum — See paragraph 27.3, Process Flavors.

:priority — See paragraph 27.3, Process Flavors.

process-run-restartable-function *name-or-keywords function* &rest *args* Function

This function is the same as **process-run-function** except that the default for resetting or warm booting is to restart the process. You can produce the same effect by using **process-run-function** with **:restart-after-reset** set to **t** and **:warm-boot-action** set to **'sys:process-warm-boot-restart**.

Process Flavors

27.3 Process flavors are the flavors of process provided by the system. You can define additional flavors of your own, provided that you include one of these basic flavors.

sys:process Flavor

This flavor is the basic flavor. The allowable instance variables are as follows:

name
stack-group
runstate
initial-form
run-reasons
wait-argument-list
wait-function
initial-stack-group
quantum
arrest-reasons
priority
warm-boot-action

You can also specify all possible initialization keywords used by **make-stack-group** because an initial stack group is created, unless you provide one with the **:initial-stack-group** option.

sys:simple-process Flavor

A simple process is not a process in the conventional sense, for it has no stack group of its own. Instead of having a stack group that is resumed when it is time for the process to run, it has a function that is called at that time. When the wait function of a simple process becomes true and the scheduler notices it, the simple process's function is called in the scheduler's own stack group. Because a simple process does not have any stack group of its own, it cannot save control state in between calls; any state that it saves must be saved in a data structure.

The only advantage of simple processes over normal processes is that they use up less system overhead because they can be scheduled without the cost of resuming stack groups. They are intended as a special, efficient mechanism for certain purposes. For example, packets received from the Chaosnet are examined and distributed to the proper receiver by a simple process that wakes up whenever there are any packets in the input buffer.

However, simple processes are harder to use because you cannot save state information across scheduling. That is, when the simple process is ready to wait again, it must return or it can call **process-wait**, which is equivalent to a throw to the scheduler.

Another drawback to simple processes is that if the function signals an error, the scheduler itself may be broken and multiprocessing then stops; this situation can be difficult to repair. Also, while a simple process is running, no other process is scheduled, so simple processes should never run for a long time without returning.

Asking for the stack group of a simple process does not signal an error but returns the process's function instead.

Process Generic Operations

27.4 Process generic are the operations are defined on all flavors of process. Certain process flavors can define additional operations. Not all possible operations are listed here, only those of interest to the user.

:name Method of **sys:process**

This method returns the name of the process, which was the first argument to **make-process** or **process-run-function** when the process was created. The name is a string that appears in the printed representation of the process, stands for the process in the **peek** display, and so on.

:stack-group Method of **sys:process**

This method returns the stack group currently executing on behalf of this process. This can be different from the initial stack group if the process contains several stack groups that coroutine among themselves or if the process is in the error handler, which runs in its own stack group.

Note that the stack group of a *simple* process is not a stack group at all, but a function. (For more information on simple processes, see paragraph 27.3, Process Flavors.)

:initial-stack-group Method of **sys:process**

This method returns the stack group in which the initial function is called when the process starts up or is reset.

:initial-form Method of **sys:process**

This method returns the initial form of the process. However, this is not actually a Lisp form; it is a cons whose car is the initial function and whose cdr is the list of arguments to which that function is applied when the process starts up or is reset.

In a simple process, the initial form is a list of one element: the process's function. (For more information on simple processes, see paragraph 27.3, Process Flavors.)

To change the initial form, use the **:preset** operation.

:wait-function Method of **sys:process**

This method returns the process's current wait function, which is the predicate used by the scheduler to determine if the process is capable of running. This is **#'true** if the process is running and **#'false** if the process has no current computation (for instance, if it has just been created), if its initial function has returned, or if the process has been *flushed*.

:wait-argument-list Method of **sys:process**

This method returns the arguments to the process's current wait function. This returned value is frequently the &rest argument to **process-wait** in the process's stack, rather than a true list.

:whostate Method of **sys:process**

This method returns a string that is the state of the process to appear in the status line at the bottom of the screen. This method displays `Run` if the process is running or trying to run; otherwise, it displays the reason why the process is waiting. If the process is stopped, then this run-state string is ignored and the status line displays `Arrest` if the process is arrested or `Stop` if the process has no run reasons.

:quantum Method of **sys:process**
:set-quantum *60ths* Method of **sys:process**

These methods return or change the number of 60ths of a second this process is allowed to run without waiting before the scheduler runs another process. The quantum defaults to 60, that is, 1 second.

:quantum-remaining Method of **sys:process**

This method returns the amount of time remaining for this process to run in the current time slice, in 60ths of a second.

:priority Method of **sys:process**
:set-priority *priority-number* Method of **sys:process**

These methods return or change the priority of this process. The larger the number, the longer this process is allowed to run. Within a priority level, the scheduler runs all processes capable of running in a round-robin fashion. Regardless of priority, a process does not run for more than its quantum. The default priority is 0. Important processes such as the Chaosnet process have positive priorities. Background processes such as the hardware monitor have negative priorities. (It is unusual to use any priority other than 0.)

:warm-boot-action Method of **sys:process**
:set-warm-boot-action *action* Method of **sys:process**

These methods return or change the process's warm-boot action, which controls what happens if the machine is booted while this process is active. (Despite the name, this method applies to both cold and warm booting.) This can be either **nil** or **:flush**, meaning to flush the process, or it can be a function to call. (The **:flush** method is described later in this section.) The default is **sys:process-warm-boot-delayed-restart**, which resets the process, causing it to start over at its initial function once the scheduler is running.

You can also use **sys:process-warm-boot-reset**, which throws out of the process's computation and kills the process, or **sys:process-warm-boot-restart**, which is like the default but restarts the process at an earlier stage of system reinitialization. This action is used for processes such as the keyboard process and the Chaos background process, which are needed for reinitialization itself.

:simple-p Method of **sys:process**

This method returns **nil** for a normal process, or true for a simple process. (For more information on simple processes, see paragraph 27.3, Process Flavors.)

:run-reasons Method of **sys:process**

This method returns the list of run reasons, which are the reasons why this process should be active (allowed to run).

:run-reason *object* Method of **sys:process**

This method adds *object* to the process list of run reasons. If the process has no arrest reasons after this method is run, the process becomes active and can run if the **wait** function returns true.

:revoke-run-reason *object* Method of **sys:process**

This method removes *object* from the process's list of run reasons. The object must be **eq** to some item in the list. If the process was active and this method causes the list of run reasons to become empty, the process stops.

:arrest-reasons Method of **sys:process**

This method returns the list of arrest reasons, which are the reasons why this process should be inactive (forbidden to run).

:arrest-reason *object* Method of **sys:process**

This method adds *object* to the process's list of arrest reasons. This causes the process to become inactive and enter an arrested state.

:revoke-arrest-reason *object* Method of **sys:process**

This method removes *object* from the process's arrest reasons, and it can activate the process.

:active-p Method of **sys:process**
:runnable-p Method of **sys:process**

These two operations are the same. If the process is active (it can run if its wait function allows), true is returned. If the process is stopped, **nil** is returned.

:preset *function* &rest *args* Method of sys:**process**

Sets the process's initial function to *function* and initial arguments to *args*. The process is then reset so that it throws out of any current computation and starts itself up by applying *function* to *args*. A **:preset** operation on a stopped process returns immediately but does not activate the process because it does not have a run reason; hence, the process does not actually apply *function* to *args* until it is activated later.

:reset &optional *no-unwind kill-p* Method of sys:**process**

This method sets the **wait** function to #'true and forces the process to throw out of its present computation and apply its initial function to its initial arguments when it next runs. The throwing out is skipped if the process has no present computation (that is, if it was just created), or if the *no-unwind* option so specifies. The possible values for *no-unwind* are the following:

- **:unless-current** or **nil** — Unwinds unless the stack group to be unwound is the one that is currently being executed in or that belongs to the current process. This is the default.
- **:always** — Unwinds in all cases. This value may cause the operation to throw through its caller instead of returning.
- **t** — Never unwinds (this must be the symbol **t**, not merely a non-**nil** value).

If *kill-p* is true, the process is to be killed after it is unwound. This is for internal use by the **:kill** operation only.

A **:reset** operation on a stopped process returns immediately but does not activate the process because it does not have a run reason; hence, the process is not actually reset until it is activated later.

:flush Method of sys:**process**

Forces the process to wait forever. A process cannot **:flush** itself. Flushing a process is different from stopping it in that the process is still active, and if it is reset or preset, it starts running again.

NOTE: A flushed process is reset by the window system if it is a process controlled by **process-mixin** and the window is selected or exposed. Also, it is given a run reason if it does not have one.

:kill &optional *wait-p* Method of sys:**process**

Eliminates the process. It is reset, stopped, and removed from sys:**all-processes**. If *wait-p* is true, then the primary method waits for the process to be cleaned up and deleted before returning.

:interrupt *function* &rest *args* Method of **sys:process**

Forces the process to **apply** *function* to *args*. When *function* returns, the process continues the interrupted computation. If the process is waiting, it wakes up, calls *function*, and then waits again when *function* returns.

If the process is stopped, it does not **apply** *function* to *args* immediately but does so later when it is activated. Normally, the **:interrupt** operation returns immediately, but if the process's stack group is in an unusual internal state, **:interrupt** may have to wait for the process's stack group to exit that state.

sys:set-process-wait *simple-process wait-function wait-argument-list* Function

This function sets the *wait-function* and *wait-argument-list* of *simple-process*.

Other Process Functions

27.5 The following are miscellaneous process functions.

process-enable *process* Function

This function activates *process* by revoking all of its run and arrest reasons, then giving it a run reason of **:enable**.

process-reset-and-enable *process* Function

This function resets *process*, then enables it.

process-disable *process* Function

This function stops *process* by revoking all of its run reasons and all of its arrest reasons.

The following functions are obsolete because they simply duplicate what can be done by sending a message.

process-initial-form *process* Function

This function returns the initial form of a process, as does the **:initial-form** operation.

process-initial-stack-group *process* Function

This function returns the initial stack group of a process, as does the **:initial-stack-group** operation.

process-name *process* Function

This function returns the **name** of a process, as does the **:name** operation.

process-preset *process function* &rest *args* Function

This function sends a **:preset message**.

process-reset *process* Function

This function sends a **:reset message**.

process-stack-group *process* Function

This function returns the current stack group of a process, as does the **:stack-group** operation.

process-wait-argument-list *process* Function

This function returns the arguments to the current wait function of a process, as does the **:wait-argument-list** operation.

process-wait-function *process* Function

This function returns the current wait function of a process, as does the **:wait-function** operation.

process-whostate *process* Function

This function returns the current status line state string of a process, as does the **:whostate** operation.

The Scheduler

27.6 At any time, there is a set of active processes. Each active process is either currently running, trying to run, or waiting for a particular condition to become true. The active processes are managed by a special stack group called the *scheduler*, which repeatedly cycles through the active processes, determining for each process whether it is ready to be run or is waiting. The scheduler determines whether a process is ready to run by applying the process's wait-function to its wait-argument-list. If the wait function returns a non-**nil** value, then the process is ready to run; otherwise, it is waiting. If the process is ready to run, the scheduler resumes the current stack group of the process.

The process is now the *current process*, that is, the one process that is running on the machine. The scheduler sets the variable ***current-process*** to this process. It remains the current process and continues to run until it either decides to wait or a *sequence break* occurs. In either case, the scheduler stack group is resumed, and it continues to cycle through the active processes. In this way, each process that is ready to run gets its share of time in which to execute.

A process can wait for a particular condition to become true by calling **process-wait** (described later in this section), which sets up its wait function and wait argument list accordingly, and resumes the scheduler stack group. A process can also wait for just a moment by calling **process-allow-schedule** (described later in this section), which resumes the scheduler stack group but leaves the process able to run; it runs again as soon as all other processes capable of running have had a chance.

A sequence break is a kind of interrupt that is generated by the Lisp system for any of a variety of reasons; when it occurs, the scheduler is resumed. The function **sys:sb-on** (described later in this section) can be used to control when sequence breaks occur. The default is to perform a sequence break once a second. Thus, if a process runs continuously without waiting, it is forced to return control to the scheduler once a second so that any other processes capable of running get their turn.

The system does not generate a sequence break when a page fault occurs; thus, time spent waiting for a page to come in from the disk is *charged* to a process the same as time spent computing. That time cannot be used by other processes. Since a sequence break is not generated when a page fault occurs, the whole implementation of the process system can reside in ordinary virtual memory so that it is not overly concerned about paging. The performance penalty is small since Explorers are personal computers and are not multiplexed among a large number of processes. Usually, only one process is ready to run at a time.

A process's wait function is free to touch any data structure it chooses and to perform any computation it chooses. Of course, wait functions should be kept simple, using only a small amount of time and touching only a small number of pages. Otherwise, system performance is impacted since the wait function consumes resources even when its process is not running. Wait functions should be written in such a way that they cannot produce errors. If a wait function gets an error, the error occurs inside the scheduler. All scheduling comes to a halt, and the scheduler process is thrown into the error handler. Note that **process-wait** calls the wait function once before giving it to the scheduler, so an error due simply to bad arguments does not occur inside the scheduler.

Note that a process's wait function is executed inside the scheduler stack group, *not* inside the current stack group of the process. Thus, a wait function cannot access special variables bound in the process. It is allowed to access global variables. A wait function can also access variables bound by a process through the closure mechanism, but more commonly, any values needed by the wait function are passed to it as arguments. (For more information, see Section 17, Closures.)

inhibit-scheduling-flag Variable

The value of this variable is normally **nil**. If it is true, sequence breaks are deferred until **inhibit-scheduling-flag** becomes **nil** again. Thus, no process other than the current process can run.

current-process Variable

This is the process that is currently executing, or **nil** while the scheduler is running. When the scheduler calls a process's wait function, it binds **current-process** to the process so that the wait function can access its process.

without-interrupts {*body-form*}* Macro

The *body-forms* are evaluated with **inhibit-scheduling-flag** bound to **t**. This is the recommended way to lock out multiprocessing over a small critical section of code to prevent timing errors. In other words, the body is an *atomic operation*. The value(s) of a **without-interrupts** is/are the value(s) of the last form in the body. For example:

```
(without-interrupts
 (push item list))

(without-interrupts
 (cond ((member item list)
        (setq list (delete item list))
        t)
       (t nil)))
```

process-wait *run-state function* &rest *arguments* Function

This function is the primitive for waiting. The current process waits until the application of *function* to *arguments* returns non-**nil** (at which time **process-wait** returns). Note that *function* is applied in the environment of the scheduler, not in the environment of the **process-wait**, so bindings that were in effect when **process-wait** was called are *not* in effect when *function* is applied. Be careful when using any free references in *function*. The *run-state* argument is a string containing a brief description of the reason for waiting. If the status line at the bottom of the screen is examining this process, it shows *run-state*. For example:

```
(process-wait "Sleep"
  #'(lambda (initial-time)
               (> (time-difference (time) initial-time)
                  100.))
  (time))

(process-wait "Buffer"
  #'(lambda (b) (not (zerop (buffer-n-things b))))
  the-buffer)
```

sleep *interval* &optional *run-state* *[c]* Function

This function stops execution, waits for the approximate number of seconds specified by *interval*, and then resumes execution. You must specify a positive, noncomplex number for *interval* (not necessarily an integer). The **sleep** function returns **nil**. The *run-state* argument, which defaults to "Sleep", is a text string that is displayed in the status line.

process-sleep *interval* &optional *run-state* Function

This function simply waits for *interval* 60ths of a second and then returns **t**. It uses **process-wait**. The *run-state* argument, which defaults to "Sleep", is a text string that is displayed in the status line.

process-wait-with-timeout *whostate interval function* &rest *arguments* Function

This function resembles **process-wait** except that if *interval* 60ths of a second elapse and the application of *function* to *arguments* is still returning **nil**, then **process-wait-with-timeout** returns anyway. The value returned is the value of applying *function* to *arguments*; thus, it is non-**nil** if the wait condition has been satisfied, or it is **nil** for a time-out.

with-timeout (*interval* {*timeout-form*}*) {*body-form*}* Macro

This macro executes the *body-forms* with a timeout in effect for *interval* 60ths of a second. If the *body-forms* finish before that much time elapses, then the values of the last *body-form* are returned.

If the *body-forms* have not completed when *interval* has elapsed, their execution is terminated with a **throw** caught by the **with-timeout** macro. Then the *timeout-forms* are evaluated and the values of the last of the *timeout-forms* are returned.

For example, the following form is a convenient way to ask a question and assume a particular answer if the user does not respond promptly:

```
(with-timeout ((* 60. sixty-seconds) (format *query-io* "Y") t)
   (y-or-n-p "Really do it? (Yes if no answer in one minute)"))
```

This form is appropriate for queries likely to occur when the user has walked away from the terminal and expects an operation to finish unattended.

process-allow-schedule Function

This function simply waits momentarily; all other processes get a chance to run before the current process runs again.

sys:scheduler-stack-group Variable

This variable is the stack group in which the scheduler executes.

sys:clock-function-list Variable

This variable is a list of functions to be called by the scheduler 60 times a second. Each function is passed one argument: the number of 60ths of a second after the last time that the functions on this list were called. These functions implement various system overhead operations, such as blinking the blinking cursor on the screen. Note that these functions are called inside the scheduler, as are the functions of simple processes (described later in this section).

The scheduler calls these functions as often as possible but never more often than 60 times a second. That is, if there are no processes ready to run, the scheduler calls the functions 60 times a second, assuming that, collectively, they take less than one 60th of a second to run. If there are processes continually ready to run, then the scheduler calls these functions as often as it can, usually once a second, since that is how often the scheduler gets control.

sys:active-processes Variable

This variable is the scheduler's data structure. It is a list of lists, where the car of each element is an active process or **nil** and the cdr is information about that process.

sys:all-processes Variable

This variable is a list of all the processes in existence. It is used mainly for debugging.

sys:initial-process Variable

This variable is the process in which the system starts up when it is booted.

sys:sb-on &optional *when* Function

The **sys:sb-on** function controls which events cause a sequence break, such as when rescheduling occurs. The following are the possible values for *when*:

:clock — This event happens periodically at intervals measured by a clock. The clock forces a sequence break on every tick (one second intervals), causing rescheduling to occur frequently. Therefore, **:clock** forces sequence breaks often enough without the need for breaks from either **:keyboard** or **:chaos**. If **:clock** is not enabled, however, the following two keywords should be.

:keyboard — This event happens when a character is received from the keyboard.

:chaos — This event happens when a packet is received from the Chaosnet or when transmission of a packet to the Chaosnet is completed.

Because the keyboard and Chaosnet are heavily buffered, there is no particular advantage to enabling the **:keyboard** and **:chaos** events, unless the **:clock** event is disabled.

With no argument, **sys:sb-on** returns a list of keywords for the currently enabled events.

With an argument, the set of enabled events is changed. The argument can be a keyword, a list of keywords, **nil** (which disables sequence breaks entirely since it is the empty list), or a number, which is the internal mask and is not documented here.

Locks

27.7 A *lock* is a software construct used for synchronization of two processes. A lock is either held by a process or is free. When a process tries to seize a lock, it waits until the lock is free, and then it becomes the process holding the lock. When it is finished, it unlocks the lock, allowing another process to seize it. When used this way, a lock protects a resource or data structure so that only one process at a time can use it.

In the Explorer system, a lock is a locative pointer to a cell. If the lock is free, the cell contains **nil**; otherwise, it contains the process that holds the lock. The **process-lock** and **process-unlock** functions are written in such a way as to guarantee that two processes can never both think that they hold a certain lock; only one process can ever hold a lock at one time.

Creating locks is easy. The locative can point to any cell; however, a symbol-value cell works fine. For example:

```
(defvar resource-lock nil)       ; lock value is stored here
(defvar key-to-the-resource
    (locf resource-lock))
 ...<use the resource>...
(process-unlock key-to-the-resource)
```

See Section 29 on locatives for more information.

process-lock *locative* &optional *lock-value run-state time-out* Function

This function is used to seize the lock that *locative* points to. If necessary, **process-lock** waits until the lock becomes free. When **process-lock** returns, the lock has been seized. The *lock-value* argument indicates the object to store into the cell specified by *locative* (which cannot be **nil** and defaults to ***current-process***), and *run-state*, which defaults to "Lock", is passed on to **process-wait**.

If *time-out* is non-nil, it should be a fixnum representing a time interval in 60ths of a second. If it is necessary to wait longer than this interval, an error with condition name **sys:lock-timeout** is signaled.

process-unlock *locative* &optional *lock-value* Function

This function is used to unlock *locative*. The unlocking takes place if the lock pointed to by *locative* contains *lock-value*, which defaults to ***current-process***. If the value of *locative* is the unlocked state of **nil** or is something other than ***current-process***, an error is signaled. Otherwise, the lock is unlocked. The *lock-value* argument must have the same value as the *lock-value* parameter to the matching call to **process-lock**, or an error is signaled. The default value for *lock-value* is **current-process**.

It is a good idea to use **unwind-protect** to make sure that you unlock any lock that you seize. For example, suppose you write the following:

```
(unwind-protect
  (progn (process-lock lock-3)
          (function-1)
          (function-2))
  (process-unlock lock-3))
```

Then, even if `function-1` or `function-2` performs a **throw**, `lock-3` is unlocked correctly. Particular programs that use locks often define macros that combine this **unwind-protect** into a convenient stylistic device.

with-lock (*lock* &key **:norecursive**) {*body-form*}* Macro

This macro executes the *body-form* with *lock* locked. The *lock* argument should actually be an expression whose value is the status of the lock; it is used inside **locf** to obtain a locative pointer with which the locking and unlocking are performed. It is permissible for one process to lock a lock multiple times, recursively, using **with-lock**. However, if **:norecursive** is given a value of **t**, then a recursive call generates an error.

The value should be literally one of the symbols **t** or **nil**; they are interpreted at macro expansion time, not at run time. Only one keyword is allowed.

store-conditional *location oldvalue newvalue* Function

This function stores *newvalue* into *location* if and only if *location* currently contains *oldvalue*. The returned value is true if and only if the cell was changed.

If *location* is a list, the cdr of the list is tested and stored in. This acts in accordance with the general principle of how to access the contents of a locative properly and makes the following code work:

```
(store-conditional (locf (cdr x))...)
```

An even lower-level construct is the subprimitive **sys:%store-conditional**, which is like **store-conditional** with no error-checking.

INITIALIZATIONS

28

Introduction

28.1 An application or subsystem often must synchronize an aspect of its work with an event that occurs outside of its immediate control. For example, when the Explorer system boots, the file system wants to set up access to the local file band. The initialization software provides hooks within the system that notify applications when such events occur. Two key elements are the initialization form supplied by the application and the event to which the initialization is to be tied, which is selected by the application from a predefined set of events.

The Explorer system uses initialization lists to organize and track the status of all the initialization routines that need to be run. Each list contains initialization forms whose evaluation is triggered by an event in the system. Several predefined initialization lists, which correspond to events that happen during system processing, are supported. An application can define other initialization-lists that have evaluations triggered by an event defined by the application.

Initialization forms are added to lists incrementally. Thus, as applications are added to the system, the associated initialization forms are added to the appropriate initialization lists. This allows the initialization code to reside with the source code of the application rather than being built into the system. When the initializations are run, the forms are evaluated in the order they were added to the list, so the precedence is set when the application is loaded.

Each initialization has four attributes:

- Name — A string that names the initialization
- Form — The Lisp form to be evaluated
- Flag — The indicator of whether the form has been evaluated
- Source — The source file (if any) for the initialization

At the appointed time, the initializations are evaluated in the order that they were added to the initialization list.

Initialization Keywords

28.2 Two sets of keywords are used to support the initialization functions. A separate set of keywords is used to denote the time when an initialization form is run. The keywords in Table 28-1 identify the various initialization lists. Table 28-2 identifies keywords used to denote initializaiton time.

Table 28-1 Initialization List Keywords

Keyword	Description
:once	Identifies the initialization list **sys:once-only-initialization-list**. This list contains initializations that need to be performed only once when the subsystem is loaded and must never be done again. For example, some databases need to be initialized the first time a subsystem is loaded but should not be reinitialized every time a new version of the software is loaded into a currently running system. This list is for such a situation. The **initializations** function never sees this list: its *when* keyword defaults to **:first**, so normally the form is evaluated only at load time and only if it has not been evaluated already.
:system	Identifies the initialization list **sys:system-initialization-list**. This list is for items that need to be performed before other initializations can work. Included in this category are initializing the process and window systems, the file system, and the network. The initializations on this list are run every time the machine is cold or warm booted, as well as when the initialization is initially added to the list, unless explicitly overridden by a **:normal** option in the keywords list. In general, the system list should not be touched by user subsystems, although you may occasionally need to do so.
:cold	Identifies the initialization list **sys:cold-initialization-list**. This list is used for items that must be run once at cold-boot time. The initializations on this list are run after the ones on the *system* list but before the ones on the *warm* list. They are run only once but are reset by **disk-save**, thus giving the appearance of being run only at cold-boot time.
:warm	Identifies the initialization list **sys:warm-initialization-list**. This list is used for items that must be run whenever the machine is booted, including warm boots. The function that prints the greeting, for example, is on this list. Unlike the *cold* list, the *warm* list initializations are run regardless of their flags.
:user-application	Identifies the initialization list **sys:user-application-initialization-list**. This list is used for items that must be run after all the other system initializations have been run but before entering the read-eval-print-loop in the Lisp Listener. This list should contain user application initializations needed prior to user interaction.
:before-cold	Identifies the initialization list **sys:before-cold-initialization list**. This list is a variant of the *cold* list. These initializations are run before the partition is saved by **disk-save**. They prepare the environment for a cold boot by performing such actions as logging off the user and dismounting the file system. They happen only once when the partition is saved, not each time it is started up.

Table 28-1 Initialization List Keywords (Continued)

Keyword	Description
:site :site-option	Identifies the initialization list **sys:site-option-initialization-list**. This list is run every time changes are made to an item that affect the network configuration of the site as a whole. Again, you should have no occasion to invoke this initialization list because it is usually invoked automatically as part of the process of configuring the machine for communication on a network.
:full-gc	Identifies the initialization list **sys:full-gc-initialization-list**. This list is run by the **full-gc** function immediately before garbage collecting. Initializations can be put on this list to discard pointers to bulky objects or to turn lists into cdr-coded form so that they remain permanently localized.
:login	Identifies the initialization list **sys:login-initialization-list**. The **login** function runs the *login* list.
:logout	Identifies the initialization list **sys:logout-initialization-list**. The **logout** function runs the *logout* list.

These initialization lists are processed in the following order:

1. System initialization list
2. Cold initialization list
3. Warm initialization list
4. User application initialization list

User applications are free to create their own initialization lists to be run at their own times. Some system programs, such as the editor, have their own initialization list for their own purposes. See the **sys:initialization-keywords** (paragraph 28.3, Lisp Forms Associated With Initializations) variable for more information.

Table 28-2 Initialization Time of Execution Keywords

Keyword	Description
:normal	Initialization forms should not be evaluated until the time comes for this type of initialization. This keyword is the default, unless the **:system** or **:once** initialization list is specified.
:now	Evaluates the initialization form now as well as adding it to the list.
:first	Evaluates the initialization form now if it is not flagged as having already been evaluated before. This keyword is the default if **:system** or **:once** is specified.
:redo	Does not evaluate the initialization now; also sets the status flag to indicate that it has not been run even if the initialization is already in the list and flagged as having been run.
:head-of-list	Causes the initialization to be placed at the front of the initialization list instead of at the end, which is the default.

Lisp Forms Associated With Initializations

28.3 The following Lisp forms are associated with initializations.

add-initialization *name form* &optional *keywords initialization-list-name* Function

This function adds an initialization called *name* with the specified *form* to the initialization list specified either by *initialization-list-name* or by a keyword. The *name* argument can be a string or a symbol. If the initialization list already contains an initialization called *name*, this function changes its form to *form*.

The *initialization-list-name* argument, if specified, is a symbol that has as its value the initialization list. If it is unbound, it is initialized to **nil** and is given a **sys:initialization-list** property of **t**. If a keyword specifies an initialization list, *initialization-list-name* is ignored and should not be specified. The default is the system warm initialization list.

Two types of keywords are allowed in *keywords*. The first type (those in Table 28-1) specifies which initialization list to use. The second type (those in Table 28-2) specifies when to evaluate the *form*. Although the symbols in this list are elements from the KEYWORD package, they are not keywords in the sense that they must be followed by a value. Each of these keywords is *assertive* in nature; that is, its mere presence implies a true value. If keywords from both tables are used, the keyword that identifies the list should precede the keyword that indicates when the list should be run in the list of *keywords*. Otherwise, the keyword identifying a list may override the keyword identifying when the list is run.

The **add-initialization** function keeps each list ordered so that initializations added first are at the front of the list. Therefore, by controlling the order of execution by the order of additions, you can control explicit dependencies on the order of initialization. Typically, the order of additions is controlled by the loading order of files. The system list is the most crucially ordered of the predefined lists.

sys:initialization-keywords Variable

Each element on this list defines the keyword for one initialization list. Each element is a list of two or three symbols. The first is the keyword symbol that names the initialization list. The second is a special variable, having as a value the initialization list itself. The third, if present, is a keyword defining the default *time* at which initializations added to the list should be evaluated. This keyword should be from Table 28-2. The third element acts as a default value if the call to **add-initialization** fails to identify when the initialization should be run. If the third element is not present, the default value **sys:normal** is assumed.

delete-initialization *name* &optional *keywords initialization-list-name* Function

This function removes the specified initialization from the specified initialization list. The *name* argument should be a string or symbol. The *keyword* argument can be a keyword or a list of them. The only meaningful action you can take with this argument is to identify which initialization list should be used. If *keyword* is **nil**, then you should supply the symbol name of the initialization list as the *initialization-list-name* argument, which defaults to **'sys:warm-initialization-list.**

initializations *initialization-list-name* &optional *redo-flag-p flag-value-p* Function

This function performs the initializations in the specified list. The *redo-flag-p* argument controls whether initializations already performed are performed again: the default value of **nil** means no, and a non-**nil** value means yes. The *flag-value-p* argument is the value to be placed in the flag slot of an entry. If unspecified, this argument defaults to **t**, meaning that the system should remember that the initialization has been performed.

reset-initializations *initialization-list-name* Function

This function changes the status flag of all entries in the specified list to indicate that they have not been run. The next time **initializations** with this list is called, all of the forms will be evaluated. The *initialization-list-name* argument should be a symbol.

Adding Initializations for Applications

28.4 The following procedures use a fictitious application named `gripper` that initializes a data array whenever the system is booted. You have two options for having the system initialize `gripper`. One way is to add `gripper`'s initialization to the:**user-application** initialization list.

Adding `gripper`'s initialization to the **:user-application** initialization list requires only that the application-installation procedures executes the **add-initialization** function:

```
(add-initialization "Gripper Array Initializations"
                    '(initialize-gripper-data-array) :user-application)
```

In this example, `initialize-gripper-data-array` is an application defined for creating a data array for the `gripper` application. Because the **:user-application** keyword is specified, the `gripper` application will be initialized after the system initializations stabilize the system. By specifying the **:now** keyword in addition to **:user-application,** you could have initialized `gripper` immediately.

```
(add-initialization "Gripper Array Initializations"
          '(initialize-gripper-data-array) '(:user-application :now))
```

Suppose, however, that you want to initialize `gripper` at a time other than those supplied by the Explorer system. To do this, you must create your own `gripper` initialization list.

Creating your own initialization list requires several steps. First, you should define a variable identifying the `gripper` initialization list:

```
(defvar gripper-initialization-list nil
"This variable holds all the initializations for the gripper application.")
```

Although this symbol would be created automatically for you by the **add-initialization** function, it is better coding style to have it reside in a source file and to provide it with a documentation string so that META-. will have a reference for this variable. The list is originally empty, but as you add initializations to it, they are consed onto the list. Now that the new list exists, add the forms necessary to perform the initialization by executing the **add-initialization** function:

```
(add-initialization "Gripper Array Initializations"
                 '(initialize-gripper-data-array) nil
                  'gripper-initialization-list)     ; Specifies the
                                                     ; gripper list.
```

As with the previous example, `initialize-gripper-data-array` is a predefined function for creating the data array for the gripper application.

In this example, you have specified the actual name of the initialization list `gripper-initialization-list` rather than using an assigned keyword that represents it.

Although it is optional, you can assign a keyword to the name of your application's initialization list. Doing so is merely a matter of convenience. For example, instead of specifying the full `gripper-initialization-list` name as the value of the *initialization-list-name* parameter of **add-initialization**, you can assign a keyword to the name. Then, when you invoke the function, you need to specify nothing for the *initialization-list-name* parameter. Instead, use the `:gripper` keyword just as you used the `:user-application` keyword in the first example of this paragraph.

To add your own initialization keyword, you need only attach the keyword of your choice (`:gripper` in this case) to the list of initialization keywords:

```
(push '(:gripper gripper-initialization-list sys:normal)
                         sys:initialization-keywords)
```

At this point, you have an initialization list for the `gripper` application, and you have added the necessary forms to that list. However, you must still execute the initialization.

Assuming that you are not using a predefined initialization list, you must execute the function **initializations** to actually evaluate the forms on the list that you have created. For this example, assume that you want to initialize the `gripper` application at the time it is invoked. If this is the case, your function definition for `gripper` would begin similar to the following:

```
(defun gripper '
   (initializations 'gripper-initialization-list)
                           .
                           .
                           .
```

LOCATIVES

29

Introduction

29.1 The data type **locative** defines a Lisp object used as a *pointer* to a *cell*. Locatives are inherently a more low-level construct than most Lisp objects: they require some knowledge of the nature of the Lisp implementation.

A *cell* is a machine word that can hold a (pointer to a) Lisp object. For example, a symbol has five cells: the print name cell, the value cell, the function cell, the property list cell, and the package cell. The value cell holds (a pointer to) the binding of the symbol, and so on. Also, an array leader of length *n* has *n* cells, and an **art-q** array of *n* elements has *n* cells. (Numeric arrays do not have cells in this sense.) A locative is an object that points to a cell; it lets you refer to a cell so that you can examine or alter its contents.

Functions That Return Locatives

29.2 The following functions, special forms, and macros return locatives.

locf *place* Macro

This macro takes a *place* form (a form that accesses a cell) and produces a corresponding form to create a locative pointer to that cell; in this sense, it is analogous to **setf**. Note the following equivalence:

```
(locf a) <=> (variable-location a)
```

value-cell-location *symbol* Function

This function returns a locative pointer to the value cell of the symbol that is the value of *symbol*; *symbol* is evaluated. This is actually the internal value cell; there can also be an external value cell if the variable is closed over.

variable-location *symbol* Special Form

This special form returns a locative pointer into the value cell of which the value of *symbol* is stored. This form does not evaluate its argument, so the name of the symbol must appear explicitly in the code.

With ordinary special variables (nonconstants and nonsystem-defined variables), this form is equivalent to the following form:

```
(value-cell-location 'symbol)
```

The compiler does not always store the values of variables in the value cell of symbols. The compiler handles **variable-location** by producing code that returns a locative to the cell where the value is actually being kept. For a local variable, this locative is a pointer into the function's stack frame. For a flavor instance variable, this locative is a pointer into the instance that is the value of **self**.

In addition, if *sym* is a special variable that is closed over by a dynamic closure, the value returned is an external value cell, the same as the value of **locate-in-closure** applied to the proper closure and *sym*. This cell always contains the value that is current only while inside the closure.

function-cell-location *symbol* Function

This function returns a locative pointer to the function cell of *symbol*.

property-cell-location *symbol* Function

This function returns a locative pointer to the location of the property cell of *symbol*. This locative pointer can be passed to **get** or **putprop** with the same results as if *symbol* itself had been passed. It is preferable to write the following:

```
(locf (symbol-plist symbol))
```

car-location *cons* Function

This function returns a locative pointer to the cell containing the car of *cons*. The argument must be a cons.

Note that there is no **cdr-location** function because of cdr-coding (see paragraph 6.2, Cdr-Coding). Instead, the cons itself serves as a locative to its cdr.

aloc *array* &rest *subscripts* Function

This function returns a locative pointer to the element cell of *array* selected by the *subscripts*. The *subscripts* must be fixnums, and their number must match the rank of *array*. Consider the following equivalence:

```
(locf (aref some-array index)) <=> (aloc some-array index)
```

ap-leader *array i* Function

The argument *array* should be an array with a leader, and *i* should be a fixnum. This function returns a locative pointer to the *i*th element of the leader of *array*. Note the following equivalence:

```
(ap-leader array index) <=> (locf (array-leader array index))
```

Functions That Operate on Locatives

29.3 The following functions operate on locatives.

locativep *object* Function

This function returns true if *object* is a locative; otherwise, it returns **nil**.

contents *locative* Function

The function **contents** returns the contents of the cell to which the locative points. To modify the contents of the cell, use **setf** with **contents**:

```
(setf (contents loc) newvalue)
```

location-boundp *locative* Function

This function returns **t** if the cell to which *locative* points contains anything except a void marker.

The void marker is a special data type, **dtp-null**, which is stored in cells to indicate that their value is missing. For example, an unbound variable actually has a void marker in the value cell. Note the following equivalence:

```
(location-boundp (locf-x)) <=> (variable-boundp x)
```

location-makunbound *locative* &optional *pointer* Function

This function stores an empty marker into the cell to which *locative* points. This cell consists of a data-type field **dtp-null** and a pointer copied from *pointer*.

The pointer field of the void marker is used to tell the error handler which variable was unbound. In the case of a symbol's value cell or function cell, it should point to the symbol header. In the case of a flavor method, it should point to the beginning of the block of data that holds the definition, which is a word containing the method's function spec.

If the second argument is not specified, then the place at which the void marker points is not defined.

Mixing Locatives With Lists

29.4 The functions **car** and **cdr** can be given a locative and return the contents of the cell at which the locative points. These two functions are equivalent to **contents** when the argument is a locative.

Similarly, the functions **rplaca** and **rplacd** can be used to store an object into the cell at which a locative points. The following three forms are equivalent:

```
(rplaca locative y) <=> (rplacd locative y) <=> (setf (contents locative) y)
```

If you are just using locatives, you should use **contents** rather than **car** or **cdr**. But you can also mix locatives and conses. For example, the same variable may usefully sometimes have a locative as its value and sometimes a cons. In such a case, it is useful that **car** and **cdr** work on locatives, and it also matters which one you use. Pick the one that is proper for the case of a cons.

For example, the following function conses up a list by adding onto the end of the list.

```
(defun simplified-version-of-mapcar (fcn lst)
    (do* ((lst lst (cdr lst))
          (result nil)
          (loc (locf result)))
         ((null lst) result)
      (setf (cdr loc)
            (setq loc (cons (funcall fcn (car lst)) nil)))))
```

The first time through the loop, `loc` points to `nil`. In subsequent iterations of the loop, `loc` points to the last cons cell in the list. Each time through the loop, the current value of `loc` is first saved as the *place* argument for the form `(setf (cdr loc) ...)`; then `loc` is changed to point to a new cons cell whose cdr is `nil`. The last action performed by the loop is the actual `setf` operation. In the general case, it replaces the cdr of the old last cons cell (pointed to by `loc` when the loop started) to point to the new cons cell, which is now being pointed to by `loc`. Note that the first time through the loop, the form `(setf (cdr loc) ...)` is actually setting the value of the variable `result` to the initial cons cell; once set, the value of `result` never changes because it is pointing to the first cons cell.

In this example, `cdr` is used rather than **contents** because the normal case is that the argument is a list.

ZETALISP COMPATIBILITY

A

Zetalisp Definitions

A.1 The symbols that make up the Zetalisp compatibilty (ZLC) package fall into two categories: external symbols and internal/incompatible symbols. Both categories are considered obsolete to the Common Lisp environment, and their use is discouraged.

Any ZLC symbol can be explicitly referenced with a ZLC package prefix in front of the symbol name. However, to permanently add the ZLC symbols to the current namespace, the current package must use the ZLC package.

Of the ZLC symbols, some have a name conflict with a corresponding symbol in the LISP package. To allow you to use both the LISP and ZLC packages at the same time, the incompatible ZLC symbols are maintained as internal symbols. Depending on the mode you are using, the Lisp Reader makes the appropriate symbol substitution to access the LISP or ZLC symbol. If you are using Common Lisp mode, you access the external symbol in the LISP package, whereas in Zetalisp mode the Reader substitutes the internal ZLC symbol of the same name. Section 1, Introduction, describes the Lisp modes and how to switch between them.

Some ZLC symbols are considered obsolete even in the latest version of Zetalisp. The use of such symbols generates compiler warnings. These warnings are made in case you are trying to generate portable Zetalisp code.

External Zetalisp Symbols

A.1.1 The following ZLC symbols are declared external.

add1 *number* Function

This function is exactly the same as **1+** in Section 3, Numbers.

adjust-array-size *array new-size* Function

This is the obsolete function for changing an array's dimensions. It is limited in that it does not accept keyword arguments as the Common Lisp function **adjust-array** does. If *array* is a one-dimensional array, its size is changed to *new-size*. If *array* has more than one dimension, its size (**array-length**) is changed to *new-size* by changing only the last dimension.

If *array* is made smaller, the extra elements are lost; if *array* is made bigger, the new elements are initialized in the same fashion as **make-array**. Consider the following examples:

```
(setq a (make-array 5))
(aset 'foo a 4)
(aref a 4) => foo
(adjust-array-size a 2)
(aref a 4) => Error
```

If the size of the array is being increased, **adjust-array-size** may have to allocate a new array somewhere. In that case, it alters *array* so that references to it are made to the new array instead. The **adjust-array-size** function returns the new array if it creates one, and otherwise it returns *array*. Be careful to be consistent about using the returned result of **adjust-array-size**, because you may end up with two arrays that are not the same (not **eq**) but share the same contents.

all-special-switch Variable

If this variable is non-**nil**, the compiler regards all variables as special, regardless of how they were declared. This variable provides compatibility with the Zetalisp mode of the interpreter at the cost of efficiency. The default is **nil**.

allow-variables-in-function-position-switch Variable

If this variable is non-**nil**, the compiler allows the use of the name of a variable in function position to mean that the variable's value should have **funcall** called on it. This variable is for compatibility with old MacLisp programs. The default value of the variable is **nil**.

%args-info *function* Function
args-info *function* Function

These functions return a fixnum called the *numeric argument descriptor* of the *function*, which describes the way the function takes arguments. The *function* argument can be a function or a function spec.

The information is stored in various bits and byte fields in the fixnum, which are referenced by the symbolic names shown below. By the usual Explorer convention, those starting with a single **%** are bit masks (meant to have **logand** or **bit-test** invoked on them with the number), and those starting with **%%** are byte specifiers, meant to be used with **ldb** or **ldb-test**.

Byte Fields of the Numeric Argument Descriptor

sys:%%arg-desc-min-args — This is the minimum number of arguments that can be passed to this function, that is, the number of required parameters.

sys:%%arg-desc-max-args — This is the maximum number of arguments that can be passed to this function, that is, the sum of the number of required parameters and the number of optional parameters. If there is an &rest argument, this is not really the maximum number of arguments that can be passed; an arbitrarily large number of arguments is permitted, subject to limitations on the maximum size of a stack frame (about 200 words).

sys:%arg-desc-evaled-rest — If this bit is set, the function has an &rest argument, and it is not quoted.

sys:%arg-desc-quoted-rest — If this bit is set, the function has an &rest argument, and it is quoted. Most special forms have this bit.

sys:%arg-desc-fef-quote-hair — If this bit is set, there are some quoted arguments other than the &rest argument (if any). The argument list should be consulted. This is only for special forms.

sys:%arg-desc-interpreted — This function is not a compiled-code object, and a numeric argument descriptor cannot be computed. The **args-info** function does not return this bit, although **%args-info** used to.

sys:%arg-desc-fef-bind-hair — This deals with argument initialization. The argument list should be consulted.

Note that **sys:%arg-desc-quoted-rest** and **sys:%arg-desc-evaled-rest** cannot both be set.

See the **sys:args-desc** function for a more efficient way to obtain this information.

array-grow *array* &rest *dimensions* Function

This function is equivalent to the following form:

(**adjust-array** *array dimensions*)

array-index-order Constant

This constant is true in more recent system versions that store arrays in row-major order (last subscript varies faster). It is **nil** in older Zetalisp system versions that store arrays in column-major order.

array-length *array* Function

This function is exactly like **array-total-size** in Section 7, Arrays.

array-pop *array* Function

This function is exactly like **vector-pop** in Section 7, Arrays.

array-push *array x* Function

This function is like **vector-push** in Section 7, Arrays, but **vector-push** is preferable because it takes arguments in an order like that of the **push** function.

array-push-extend *array x* &optional *extension* Function

This function is like **vector-push-extend** in Section 7, Arrays, but **vector-push-extend** is preferable because it takes arguments in an order like the **push** function.

ar-2-reverse *array horizontal-index vertical-index* Function

This function returns the component of *array* at *horizontal-index* and *vertical-index*. This function was intended to be used for screen pixel arrays in which the *horizontal-index* argument is used as the subscript in whichever dimension varies faster through memory. The **ar-2-reverse** function is like the Common Lisp function **aref** except that the index arguments are reversed.

aset *x array* &rest *subscripts* Function

This function stores *x* into the element of *array* selected by the *subscripts*. Note the following equivalence:

(**apply 'aset** *value array index-list*) <=>
(**setf** (**apply 'aref** *array index-list*) *value*)

Common Lisp uses the macro **setf** for storing values into arrays.

ass *predicate item a-list* Function

This function is the same as **assq**, except that it takes an extra argument that should be a predicate of two arguments. This extra argument is used for the comparison instead of **eq**. The form `(ass #'eq a b)` is the same as `(assq a b)`. You can use noncommutative predicates; the first argument to the predicate is *item*, and the second is the key of the element of *a-list*.

The Common Lisp equivalent of `(ass p x y)` is `(assoc x y :test p)`.

assq *item a-list* Function

This function looks up *item* in the association list. The returned value is the first cons whose car is **eq** to *item*, or **nil** if there is none.

The Common Lisp equivalent of `(assq x y)` is `(assoc x y :test #'eq)`.

as-1 *x array i* Function
as-2 *x array i j* Function
as-3 *x array i j k* Function

These are obsolete versions of **aset** that work for one-dimensional, two-dimensional, and three-dimensional arrays, respectively.

as-2-reverse *newvalue array horizontal-index vertical-index* Function

This function stores *newvalue* into the component of *array* at *horizontal-index* and *vertical-index*. The *horizontal-index* argument is used as the subscript in whichever dimension varies faster through memory.

Code written before the change in order of array indices can be converted by replacing calls to **make-array, array-dimension-n, aref,** and **aset** with **make-pixel-array, pixel-array-width, pixel-array-height, ar-2-reverse,** and **as-2-reverse**. It can then work either in old systems or in new ones. In more complicated circumstances, you can make conversion easier by writing code that tests the **array-index-order** constant described earlier in this appendix.

atan2 *y* &optional *x* Function

This function is exactly the same as the Common Lisp **atan** function. Note, however, that **atan** has a different meaning in Zetalisp mode than it does in Common Lisp mode.

base Variable

This variable is exactly the same as ***print-base***.

bit-test *x y* Macro

This function is exactly the same as **logtest** in Section 3, Numbers.

catch *form tag* Special Form

The obsolete usage (**catch** (*f x*) *symbol*) is equivalent to (**catch** '*symbol* (*f x*)).

***catch** *tag* {*body-form*}* Special Form

This function is exactly like **catch** in Section 14, Control Structures.

char≤ *char1* &rest *chars* Function
char≥ *char1* &rest *chars* Function

These functions are exactly like the Common Lisp **char<=** and **char>=** functions in Section 4, Characters.

check-arg-type *var-name type-name* &optional *type-description* Macro

This macro operates the same way as the Common Lisp macro **check-type**; see paragraph 20.2, Signaling Conditions.

copyalist *list* &optional *area* Function

This function is exactly like **copy-alist** in Section 6, Lists and List Structure.

copylist *list* &optional *area* Function

This function is exactly like **copy-list** in Section 6, Lists and List Structure.

copysymbol *sym copy-props* Function

This function is exactly like **copy-symbol** in Section 2, Symbols.

copytree *tree* &optional *area* Function

This function is exactly like **copy-tree** in Section 6, Lists and List Structure.

debugging-info *function-spec* Function

This function returns an association list of the information contained in the debug-info structure. Using this function is less efficient than using **get-debug-info-field**, described in Section 16, Functions.

declare-flavor-instance-variables (*flavor*) {*body*}* Macro

Sometimes you write a function that is not itself a method but is to be called by methods and has to be able to access the instance variables of the object **self**. The following form surrounds the function definition with a declaration of the instance variables for the specified flavor, which makes them accessible by name:

```
(declare-flavor-instance-variables (flavor-name)
   (defun function args body ...))
```

Any kind of function definition is allowed; it does not have to use **defun**.

If you call such a function when the value of **self** is an instance whose flavor does not include *flavor-name* as a component, an error is produced.

The preferred alternative to this function is **defun-method**.

defconst *variable initial-value* &optional *documentation* Macro

This macro is the obsolete name for **defparameter** (see Section 13, Declarations), and its use should be avoided because its name is misleading.

defselect {*function-spec*| Macro
(function-spec [[*default-handler*] *no-which-operations-p*])}
{(*operation lambda-list* {*form*}*)|
(*operation* . *symbol*)}+

This macro defines a function that is a select-method. This function contains a mapping table of subfunctions; when it is called, the first argument, a symbol in the KEYWORD package, called the *operation*, is looked up in the table to determine which subfunction to call. Each subfunction can take a different number of arguments and can have a different pattern of arguments. The **defselect** special form is useful for a variety of dispatching jobs. By analogy with the more general message-passing facilities described in Section 19, Flavors, the subfunctions are called *methods* and the list of arguments is sometimes called a *message*.

The *function-spec* argument is the name of the function to be defined. The *default-handler* argument is optional; it must be a symbol and is a function that is called if the select-method is called with an unknown operation. If *default-handler* is unsupplied or **nil**, then an unknown operation causes an error with condition name **sys:unclaimed-message**.

Normally, a method for the operation **:which-operations** is generated automatically, based on the set of existing methods. The **:which-operations** operation takes no arguments and returns a list of all the operations in the **defselect**. If *no-which-operations-p* is true, no **:which-operations** method is created; however, you can supply one yourself.

If *function-spec* is a symbol, and *default-handler* and *no-which-operations-p* are not supplied, then the first subform of the **defselect** may simply be *function-spec* by itself, not enclosed in a list.

The remaining subforms in a **defselect** are clauses, each defining one method. The *operation* argument is the name, or a list of names, of the operations to be handled by this clause. The *lambda-list* is a lambda list for defining a handler function defined by the *forms*; it should not include the first argument, which is the operation.

A clause can alternatively have the following format: (*operation* . *symbol*). In this case, *symbol* is the name of a function that is to be called when the *operation* is performed. It is called with the same arguments as the select-method, including the operation symbol itself.

defunp *name lambda-list* {*declaration*|*doc-string*}* *body* Macro

Usually, when a function uses **prog**, the **prog** form is the entire body of the function; for this reason, the macro **defunp** was created. This macro allows you to define functions in the manner of **defun**, but it is as if a **prog** were the first form of the function definition.

del *predicate item list* &optional *n* Function

This function is similar to **delq** but differs in that it takes an extra argument that should be a predicate of two arguments. It uses this predicate for the comparison test instead of **eq**.

The equivalent Common Lisp function is **delete**, which is not restricted to operating only on lists but on all sequences. It also uses a number of different keywords that increase its versatility.

del-if *predicate list* Function

This function is similar to **rem-if**, except that it destructively alters its list argument rather than modifying a copy.

The equivalent Common Lisp function is **delete-if**, which is not restricted to operating only on lists but on all sequences. It also uses a number of different keywords that increase its versatility.

del-if-not *predicate list* Function

This function is similar to **rem-if-not**, except that it destructively alters its list argument rather than modifying a copy.

The equivalent Common Lisp function is **delete-if-not**, which is not restricted to operating only on lists but on all sequences. It also uses a number of different keywords that increase its versatility.

deletef *file* &optional *error-p query-p* Function

This function is exactly like **delete-file** except that the arguments are taken in positional order instead of in keyword order.

delq *item list* &optional *n* Function

This function returns *list* with all elements that are **eq** to *item* removed. The argument *list* is destructively modified when instances of *item* are removed.

When the optional argument *n* is provided, only *n* occurrences of *item* are removed from *list*. If *n* is greater than the number of occurrences of *item* in *list*, then all occurrences of *item* in *list* are removed. The Common Lisp equivalent of this function is as follows:

(delete *item* **(the list** *list*) **:test #'eq :count** *n***)**

deposit-byte *number position size byte* Function

This function is like **dpb**, except that instead of using a byte specifier, the position and size are passed as separate arguments. The argument order is not analogous to that of **dpb** so that **deposit-byte** can be compatible with MacLisp.

difference *number* &rest *numbers* Function

This function performs the subtraction operation and does not negate if only one argument is provided; it simply returns the argument (the same as subtracting 0 from the argument).

do-forever {*body-form*}* Macro

This macro is like the Common Lisp **loop** form. Note that this implies that the first form cannot be a symbol if it is to be distinguished from the Explorer extended **loop** form.

do-local-external-symbols (*var* [*package* [*result-form*]]) {*declarations*}* {*body-form*}* Macro

This macro is exactly like the Common Lisp macro **do-external-symbols**.

do-named *name* ({(*var* [*init* [*step*]])}*)
(end-test {*result*}*)
{*declaration*}*
{*tag* | *statement*}*
Special Form

This form is like **do** but defines a **block** with a name explicitly specified by the programmer; this **block** is in addition to the **block** named **nil**, which every **do** defines. This makes it possible to use **return-from** to return from this **do-named** macro even from within an inner **do**. An ordinary **return** within an inner **do** would return from the inner **do** only.

The syntax of **do-named** is like **do**, except that the symbol **do-named** is immediately followed by the **block** name, which should be a symbol. The *name* argument is not evaluated. For example:

```
(do-named george ((a 1 (1+ a))
                  (d 'foo))
                 ((> a 4) 7)
   (do ((c b (cdr c)))
        ((endp c))
      ...
      (return-from george (cons b d))
      ...))
```

When the *name* of a **do-named** is **t**, this macro behaves somewhat peculiarly, and therefore the name **t** should be avoided. If the name is **nil**, this macro behaves like a regular **do**.

do*-named *name* ({(*var* [*init* [*step*]])}*)
(end-test {*result*}*)
{*declaration*}*
{*tag* | *statement*}*
Special Form

This special form offers a combination of all the features of **do*** and those of **do-named**.

sys:**evall** *form* &optional *nohook-p* — Function

This function evaluates *form* in the evaluator's *current* lexical environment and current mode (either Common Lisp or Zetalisp). It is typically used by the evaluator's definition of a special form to evaluate its arguments. See sys:***eval** (which is now preferred) in Section 16, Functions.

find-position-in-list *item list* — Function

This function searches *list* for an element that is **eq** to *item*. It is similar to **memq** but differs in that it returns the numeric index in the list where it found the first occurrence of *item*, or **nil** if none was found. Like the function **nth**, indexes are zero-based.

In Common Lisp, this operation is performed by the **position** function.

find-position-in-list-equal *item list* — Function

This function is like **find-position-in-list** but uses **equal** to compare *item* with elements of *list*. See also **position**.

fix *x* — Function

This function operates like **floor**, except that it does not return a second value (the remainder). See **floor** in Section 3, Numbers.

fixp *object* Function

This function is a predicate that performs the same operation as the **integerp** predicate of Common Lisp. The designers of Common Lisp named this predicate **integerp** because it was being confused with the predicate **fixnump**.

fixr *x* Function

This function operates like **round**, except that it does not return a second value (the remainder). See the **round** function in Section 3, Numbers.

fset *symbol definition* Function

This function stores *definition*, which can be any Lisp object, into the function cell of *symbol*. The **fset** function returns *definition*. Usually, the function **fdefine** should be used instead of **fset** to change a function definition because **fdefine** is more general. The function **fset** is a primitive that should be called directly only when necessary to bypass the additional bookkeeping performed by **fdefine**. This function is obsolete; use **setf** of **symbol-function** instead. Consider the following example:

```
(fset 'func '(lambda (x)
                (+ x 3)))

#'func => (lambda (x) (+ x 3))
(func 3) => 6
```

fset-carefully *symbol definition* &optional *force-flag* Function

This function is equivalent to the following:

(**fdefine** *symbol definition* **t** *force-flag*)

fsymeval *symbol* Function

This function is exactly like the **symbol-function** function in Section 2, Symbols.

funcall-self *operation* {*argument*}* Function
lexpr-funcall-self *operation* {*argument*}* *list-of-arguments* Function

This function is almost equivalent to the **funcall** form with **self** as the first argument. The **funcall-self** function used to be faster, but now **funcall** with **self** as an argument is just as fast. Therefore, **funcall-self** is obsolete. It should be replaced with **funcall** or **send** using **self** as the first argument.

Similarly, **lexpr-funcall-self** is also obsolete and should be replaced with **lexpr-send** using **self** as the first argument.

:get-input-buffer &optional *eof-is-error-p* Method of *streams*

This is an obsolete method similar to **:read-input-buffer** (see the *Explorer Input/Output Reference* manual). The only difference is that the third value is the number of significant elements in the buffer array, rather than a final index. If found in programs, it should be replaced with **:read-input-buffer**.

get-pname Function

This function is exactly like **symbol-name** in Section 2, Symbols.

greaterp &rest *numbers* Function

This function is exactly like > in Section 3, Numbers.

grindef {*function-spec*}* Special Form

See the **pprint-def** function in the *Explorer Tools and Utilities* manual.

ibase Variable

This variable is the older name for ***read-base*** (see the *Explorer Input/Output Reference* manual); the two are tied together so that changing the value of one changes the other also.

intern-soft *string-or-symbol* &optional *pkg* Function

This function is exactly like **find-symbol** in Section 5, Packages.

lessp &rest *numbers* Function

This function is exactly like < in Section 3, Numbers.

lexpr-funcall *fn arg* &rest *arglist* Function

This function is a synonym for **apply**; formerly, **apply** was limited to a two-argument case.

The **lexpr-funcall** function can also be used with a single argument—a list of a function and arguments to pass to it.

load-byte *number position size* Function

This function is like **ldb**, except that it does not use a byte specifier; *position* and *size* are passed as separate arguments. The argument order is not analogous to that of **ldb** so that **load-byte** can be compatible with MacLisp.

local-declare ({*declarations*}*) {*body-form*}* Macro

This macro causes each of *declarations* to be in effect locally within the *body-forms* only. This macro is to be used in Zetalisp to wrap declarations around forms such as **let**, **do**, or **defun**; the new way of doing this is to use a **declare** within the form before the first *body-form*.

make-equal-hash-table &key **:size :number-of-values :area :rehash-function :rehash-size :rehash-threshold :actual-size :hash-function :compare-function :funcallablep** Function

This function is the same as the **make-hash-table** function (see Section 11, Hash Tables) using the function **equal** as the keyword argument for **:test**.

make-pixel-array *width height* &rest *options* Function

This function is like **make-array**, except that the dimensions of the array are *width* and *height*, in whichever order is correct. The *width* argument is used as the dimension in the subscript that varies faster in memory, and *height* is used as the other dimension. The values of *options* are passed along to **make-array** to specify everything but the size of the array. Thus, this function is equivalent to the following:

(make-array (list *height width* **)** . *options***)**

make-syn-stream *stream-symbol* Function

For the Common Lisp equivalent of this function, see **make-synonym-stream** in the *Explorer Input/Output Reference* manual.

mem *predicate item list* Function

This function is like **memq**, except that it takes an extra argument that should be a predicate with two arguments. The extra argument is used for the comparison instead of **eq**. Consider the following equivalence:

```
(mem #'eq a b) <=> (memq a b)
```

Also, consider the following equivalence

```
(mem #'equal a b) <=> (member a b) <=> (member a b :test #'equal)
```

In the preceding equivalence, the second form is in Zetalisp and the third form is in Common Lisp.

The **mem** function is ordinarily used with equality predicates other than **eq** and **equal**, such as =. It can also be used with noncommutative predicates. The predicate is called with *item* as its first argument and the element of *list* as its second argument. For example, (**mem #'<** 4 *list*) finds the first element in *list* for which (< 4 x) is true; that is, it finds the first element greater than 4.

memass *predicate item a-list* Function

This function searches *a-list* exactly like **ass** but returns the portion of the list beginning with the pair containing *item*, rather than the pair itself. The form `(first (memass x y z))` equals the form `(ass x y z)`. You can use noncommutative predicates; the first argument to the predicate is *item* and the second is the key of the element of *a-list*.

memq *item list* Function

This function returns **nil** if *item* is not one of the elements of *list*. Otherwise, it returns the sublist of *list* beginning with the first occurrence of *item*. The comparison is made by the **eq** function. Because **memq** returns **nil** if it does not find anything and returns something non-**nil** if it finds something, it is often used as a predicate.

You can get the same effect as the Zetalisp form `(memq x y)` by writing in Common Lisp `(member x y :test #'eq)`.

minus *number* Function

This function is exactly like –.

multiple-value *variable-list form* Special Form

This special form is like the Common Lisp **multiple-value-setq** in Section 16, Functions, except that **nil** can be used as a variable name, which causes the corresponding value to be ignored. The Common Lisp name for this special form is also much more descriptive.

ncons *x* Function

This function conses *x* onto **nil**. Note the following equivalence:

```
(ncons x) <=> (cons x nil)
```

ncons-in-area *x area* Function

This function is like **ncons**, except that the construction process occurs within the specified area.

***nopoint** Variable

This variable is an obsolete Explorer extension and does not have a trailing * like all Common Lisp global variables. If its value is **nil**, a trailing decimal point is printed when an integer is printed in base 10. This action allows the numbers to be read back in correctly even if ***read-base*** is not 10 at the time of reading. The default value of ***nopoint** is **t**. The ***nopoint** variable has no effect if ***print-radix*** is non-**nil**.

pixel-array-height *array* Function

This function returns the extent of *array*, a two-dimensional array, in the dimension that varies slower through memory. For a screen array, this value is always the height. Because Common Lisp specifies that arrays are stored in row-major order, this function is equivalent to the following:

(**array-dimension** *array* 0)

pixel-array-width *array* Function

This function returns the extent of *array*, a two-dimensional array, in the dimension that varies faster through memory. For a screen array, this value is always the width. Because Common Lisp specifies that arrays are stored in row-major order, this function is equivalent to the following:

(**array-dimension** *array* 1)

plist *symbol* Function

This function is exactly like **symbol-plist** in Section 2, Symbols.

plus &rest *numbers* Function

This function is exactly like + in Section 3, Numbers.

probef *filename-or-stream* Function

For the Common Lisp equivalent of this function, see **probe-file** in the *Explorer Input/Output Reference* manual.

qc-file *filename* &optional *output-file load-flag in-core-flag package file-local-declarations dont-set-default-p* Function

Although this function accepts a formidable number of arguments, normally you need specify only one. The *filename* is given to the compiler, and the output of the compiler is written to a file whose name is *filename*, except with a file type of xld.

Macro definitions, **subst** definitions, and **special** declarations created during the compilation are canceled when the compilation is finished.

The optional arguments allow certain modifications to the standard procedure. The *output-file* argument lets you change where the output is written. The *package* argument lets you specify in which package the source file is to be read. Normally, the attributes list on the first line of the file specifies the package, and you need not supply this argument. The *load-flag* and *in-core-flag* arguments should always be **nil**. The *file-local-declarations* argument is for compiling multiple files as if they were one. The *dont-set-default-p* argument suppresses the changing of the default filename to *filename*, which normally occurs.

This function is like the Common Lisp function **compile-file** except that **compile-file** uses keywords for the options.

qc-file-load *filename* &optional *output-file load-flag in-core-flag package functions-defined file-local-declarations dont-set-default-p* Function

This function compiles a file and then loads the resulting xld file. The new way of doing this is to use **compile-file** with the **:load** option.

rass *predicate item a-list* Function

This function is to **rassq** as **ass** is to **assq**. That is, **rass** takes a predicate to be used instead of **eq**. You can use noncommutative predicates; the first argument to the predicate is *item*, and the second is the rest of the element of *a-list*. Common Lisp uses **rassoc** with the **:test** option.

rassq *item a-list* Function

This function is the reverse form of **assq**; it tries to find an element of *a-list* whose cdr (not car) is **eq** to *item*.

remainder *integer1 integer2* Function

This function returns the remainder of *integer1* divided by *integer2*. The arguments must be integers (fixnums or bignums).

See also the Common Lisp function **rem** (which is preferred for use in new programs) described in Section 3, Numbers.

rem-if *predicate list* Function
subset-not *predicate list* Function

These functions are the complement of **subset** and **rem-if-not**. These functions apply the *predicate* argument to every element of *list* and remove the elements of *list* for which the predicate returns a non-**nil** value. The reason this operation has two names is that **subset-not** refers to the operation's action if *list* represents a mathematical set. If *list* does not represent a mathematical set, the **rem-if** function is easier to remember because it means *remove if this condition is true*.

For the corresponding Common Lisp function, see **remove-if** in Section 9, Sequences.

rem-if-not *predicate list* Function
subset *predicate list* Function

This function is another name for **subset**. These functions apply the *predicate* argument to every element of *list* and remove the elements of *list* for which the predicate returns **nil**. The reason this operation has two names is that **subset** refers to the operation's action if *list* represents a mathematical set. If *list* does not represent a mathematical set, the **rem-if-not** function is easier to remember because it means *remove if this condition is not true*.

The equivalent Common Lisp function is **remove-if-not**, which can be used on arrays as well as lists. It also uses a number of different keywords that increase operational functionality.

remob *symbol* &optional *package* Function

For the Common Lisp equivalent, see **unintern** in Section 5, Packages. In **remob**, *package* defaults to the contents of the symbol's package cell, that is, the package it belongs to.

remq *item list* &optional *n* Function

This function removes *item* from *list* but *list* is not destructively altered; a copy is created and modified. If the optional argument *n* is specified, then only *n* items are removed from *list*.

The corresponding Common Lisp function is as follows:

(**remove** *item list* **:test #'eq :count** *n*)

Note, however, that **remove** has a different meaning in Zetalisp mode.

renamef *string-or-stream new-name* &key **:error :query** Function

See the **rename-file** function in the *Explorer Input/Output Reference* manual.

rest1 *list* Function
rest2 *list* Function
rest3 *list* Function
rest4 *list* Function

These functions extract the first, second, third, and fourth cdrs of *list*. For example:

```
(rest1 '(a b c d e f g)) => (b c d e f g)
(rest4 '(a b c d e f g)) => (e f g)
```

return-list *list* Special Form

This special form is like **return** (see Section 14, Control Structures), except that each element of *list* is returned as a separate value from the **block** that is exited.

Note the following equivalence:

```
(return-list list) <=> (return (values-list list))
```

:rewind Method of *streams*

This operation is obsolete. It is the same as **:set-pointer** with an argument of zero (see the *Explorer Input/Output Reference* manual).

run-in-maclisp-switch Variable

If this variable is non-**nil**, the compiler tries to warn the user about any constructs that do not work in MacLisp. Not all Explorer system functions that are not built into MacLisp cause warnings; only those that cannot be written in MacLisp by the user (for example, **make-array**, **value-cell-location**, and so on). Also, lambda-list keywords such as &optional and initialized **prog** variables are mentioned. This switch inhibits the warnings for obsolete MacLisp functions. The default value of this variable is **nil**.

selectq *keyform* {(*test* {*body-form*}*)}* Macro

This function is exactly like **case** in Section 14, Control Structures.

setplist *symbol list* Function

This function sets to *list* the list that represents the property list of *symbol*. This function is to be used with caution (or not at all) because property lists sometimes contain internal system properties, which are used by many helpful system functions. Also, it is inadvisable to have the property lists of two different symbols be **eq**, because the shared list structure causes unexpected effects on one symbol if **putprop** or **remprop** is executed on the other. The Common Lisp equivalent is as follows:

(**setf** (**symbol-plist** *symbol*) *list*)

string-length *string* Function

This function returns the number of characters in *string*. This value is 1 if *string* is a number or character object, the **array-active-length** if *string* is an array, or the **array-active-length** of the print name if *string* is a symbol.

The corresponding Common Lisp function is **length** if *string* is known to actually be a string.

string-nreverse *string* Function

This function returns *string* with the order of characters reversed, permanently changing the original string rather than creating a new one. If *string* is a number, it is simply returned. This function reverses a one-dimensional array of any type.

For standard Common Lisp, see **nreverse** in Section 9, Sequences.

string-reverse *string* Function

This function returns a copy of *string* with the order of characters reversed. This function reverses a one-dimensional array of any type.

For standard Common Lisp, see **reverse** in Section 9, Sequences.

string-reverse-search *key string* &optional *from to key-from key-to consider-case* Function

This function searches for the string *key* in *string*. The search proceeds in reverse order, starting from the index 1 less than *from*, and returns the index of the first character of the first instance found, or **nil** if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. In the *from* condition, the instance of *key* found is the rightmost one whose rightmost character is before the *from* character of *string*. When *from* is **nil**, the search starts at the end of *string*. The last character of *string* examined is the one at index *to*.

The arguments *key-from* and *key-to* can be used to specify the portion of *key* to be searched for, rather than all of *key*. Case and font are significant in character comparison if *consider-case* is non-**nil**.

For standard Common Lisp, use **search** with the **:from-end** option.

string-reverse-search-char *char string* &optional *from to consider-case* Function

This function searches through *string* in reverse order, starting from the index 1 less than *from* (when *from* is **nil**, this function starts at the end of *string*), and returns the index of the first character that is **char-equal** to *char*, or **nil** if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. The last (leftmost) character of *string* examined is the one at index *to*.

Case and font are significant in character comparison if *consider-case* is non-**nil**. In this case, **char=** is used for the comparison rather than **char-equal**.

For standard Common Lisp, use **position** with the **:from-end** option.

string-reverse-search-not-char *char string* &optional *from to consider-case* Function

This function is like **string-reverse-search-char** but searches for a character in *string* that is different from *char*.

For standard Common Lisp, use **position** with the **:test-not** and **:from-end** options.

string-search *key string* &optional *from to key-from key-to consider-case* Function

This function searches for the string *key* specified by *string*.

The search begins at *from*, which defaults to the beginning of *string*. The value returned is the index of the first character of the first instance of *key*, or **nil** if none is found. If *to* is non-**nil**, it is used in place of (**string-length** *string*) to limit the extent of the search.

The arguments *key-from* and *key-to* can be used to specify the portion of *key* to be searched for, rather than searching for all of *key*.

Case and font are significant in character comparison if *consider-case* is non-**nil**.

The corresponding Common Lisp function is **search**.

string-search-char *char string* &optional *from to consider-case* Function

This function searches through *string* starting at the index *from*, which defaults to the beginning, and returns the index of the first character that is **char-equal** to *char*, or **nil** if none is found.

If *to* is non-**nil**, it is used in place of (**string-length** *string*) to limit the extent of the search.

The corresponding Common Lisp function is **position**.

string-search-not-char *char string* &optional *from to consider-case* Function

This function is like **string-search-char** but searches *string* for a character different from *char*.

The corresponding Common Lisp function is **position** with the **:test-not** option.

string≤ *string1 string2* &key **:start1 :end1 :start2 :end2** Function
string≥ *string1 string2* &key **:start1 :end1 :start2 :end2** Function
string≠ *string1 string2* &key **:start1 :end1 :start2 :end2** Function

These functions are exactly like the Common Lisp functions **string<=**, **string>=**, and **string/=**, respectively, in Section 8, Strings.

subrp *object* Function

This predicate returns true if the argument is either a compiled code object or a microcoded function.

substring *string start* &optional *end area* Function

This function extracts a copied substring of *string*, starting at the character specified by *start* and going up to but not including the character specified by *end*. The arguments *start* and *end* are zero-origin coordinated. The length of the returned string is equal to *end* minus *start*. If *end* is not specified, it defaults to the length of *string*.

When the *area* argument is provided, it performs this operation in the specified area (see Section 25, Storage Management).

The corresponding Common Lisp function is **subseq**.

sub1 *number* Function

This function is exactly like **1-** in Section 3, Numbers.

swapf *place1 place2* Function

This function is exactly like **rotatef** in Section 2 except that it allows only two arguments.

symeval *symbol* Function

This function is exactly like **symbol-value** in Section 2, Symbols.

throw *form tag* Special Form

The obsolete usage of (**throw** (*f x*) *symbol*) is equivalent to (**throw** *'symbol* (*f x*)).

***throw** *tag values-form* Special Form

This function is exactly like **throw** in Section 14, Controls Structures.

times &rest *numbers* Function

This function is exactly like ***** in Section 3, Numbers.

tyo *char* &optional *stream* Function

This function is exactly like the Common Lisp function **write-char** in the *Explorer Input/Output Reference* manual.

viewf Function

For the Common Lisp equivalent of this function, see **view-file** in the *Explorer Input/Output Reference* manual.

xcons *x y* Function

This function is like **cons**, except that the order of the arguments is reversed in the returned object. Note the following equivalence:

```
(xcons a b) <=> (cons b a)
```

xcons-in-area *x y area* Function

This function is exactly like **xcons**, except that the construction process occurs within the specified area.

^ *x y* Function
^$ *x y* Function

These functions are exactly like **expt** in Section 3, Numbers.

≤ *number* &rest *numbers* Function
≥ *number* &rest *numbers* Function

These functions are exactly like the Common Lisp functions **<=** and **>=**, respectively, in Section 3, Numbers.

≠ *number* &rest *numbers* Function

This function is exactly like the Common Lisp function **/=** in Section 3, Numbers.

**** *x y* Function

This function is exactly like the **rem** function in Section 3, Numbers. Note that if this symbol name is read in Common Lisp mode, it must be protected by vertical bars or additional backslashes.

**** &rest *integers* Function

This function is exactly like the Common Lisp **gcd** function in Section 3, Numbers. Note that if this symbol name is read in Common Lisp mode, it must be protected by vertical bars or additional backslashes.

Internal/ Incompatible Symbols

A.1.2 The following ZLC symbols are internal because each has a name conflict with a symbol in the LISP package.

/ *number* &rest *more-numbers* Function

This function is like the Common Lisp **/** function except that if both arguments are integers, the result is a truncated integer instead of a rational number. Note that in Zetalisp mode, you must type the symbol name as **//**. See also **quotient**.

aref *array* &rest *subscripts* Function

This function is like the Common Lisp function **aref** except that it returns an integer rather than a character object when *array* is a string.

assoc *item a-list* Function

The Zetalisp version of the **assoc** function uses an **equal** predicate rather than **eql**, which is the Common Lisp default. Note the following equivalence:

```
(zlc::assoc x y) <=> (lisp:assoc x y :test #'equal)
```

atan *y x* Function

The Zetalisp version of **atan** returns the angle, in radians, whose tangent is *y*/*x*. The Zetalisp **atan** function *always* returns a nonnegative number between 0 and 2π, whereas the Common Lisp version returns a number between $-\pi/2$ and $\pi/2$.

character *x* Function

Note the following equivalence between the Zetalisp and Common Lisp versions of this function:

```
(zlc:character x) <=> (char-int (lisp:character x))
```

defstruct *name-and-zl-options* [*doc-string*] {*component-description*}+ Macro

This macro is like the Common Lisp **defstruct** macro in Section 10, Structures, except that the defaults for several options are different:

- The default **:conc-name** is the empty string.
- The default for **:callable-constructors** is **nil**.
- The **:alterant** option is created by default with the name **alter-***structure-name*.
- The default for **:type** is **:array**.
- If the **:predicate** option is not present, the default is to not create a predicate function.

delete *item list* &optional *count* Function

This function is exactly like the Common Lisp **delete** function except that it uses **equal** for the test and does not have any keyword options.

eval *form* &optional *nohook* Function

This function is like the Common Lisp **eval** in Section 16, Functions, except that the evaluation is performed in Zetalisp mode. Also, instead of the *nohook* option, the Common Lisp form provides for a local definitions environment argument.

every *list predicate* &optional *step-function* Function

This function returns true if *predicate* returns non-**nil** for every element of *list*; otherwise, **every** returns **nil**. If *step-function* is supplied, it is used to get the next element of *list* (rather than **cdr**).

float *number* &optional *float* Function

This function is exactly like the Common Lisp function of the same name in Section 3, Numbers, except that if the second argument is omitted, a coercion to **single-float** is done even when the argument was already a floating-point number.

format *stream control-string* &optional *args* Function

This function is like the Common Lisp **format** except that ~E, ~F, ~G, and ~X have different meanings. They are interpreted as follows:

- ~*n*E — Prints the argument rounded to *n* digits of precision with an explicit exponent.
- ~*n*F — Prints the argument rounded to *n* digits of precision.
- ~*n*G — Skips to the *n*th argument.
- ~*n*X — Skips over *n* spaces.

For example:

```
#!Z(format nil "~2E ~2F ~4G ~10X ~d" 1000.0 2000.0 3000.0 4000. 5000.)
=> "1.0e3  2000            5000"
```

intersection *list1* &rest *more-lists* Function
nintersection *list1* &rest *more-lists* Function

These Zetalisp functions are like the Common Lisp functions of the same name in Section 6 except that any number of lists can be supplied as arguments and the test predicate is always **eq**.

listp *x* Function

This function is the same as the Common Lisp function **consp** described in Section 6. The Common Lisp **listp** differs by returning true when the argument is **nil**.

make-hash-table &key **:test :size :rehash-size :rehash-threshold :number-of-values :area :rehash-function :actual-size :hash-function :compare-function :funcallablep** Function

This function is like the Common Lisp function of the same name, described in Section 11, Hash Tables, except that the **:test** defaults to **eq** instead of **eql**. Also, this Explorer extension includes the following additional keyword options:

:area — This argument specifies the area in which the hash table should be created. This argument is exactly like the **:area** option to **make-array** (see paragraph 7.2, Array Creation). This argument defaults to **nil** (that is, **default-cons-area**).

:rehash-function — This argument specifies the function to be used for rehashing when the table becomes full. This argument defaults to the rehashing function provided by the system. If you want to write your own rehashing function, you must understand all the internals of how hash tables work. Study the source code to find this information.

:actual-size — This argument specifies the exact size for the hash table. The size for hash tables used by the microcode for flavor method lookup must be a power of 2. This requirement differs from **:size** in that **:size** is rounded up to the nearest prime number, whereas **:actual-size** is used exactly as specified. The **:actual-size** argument, if specified, overrides **:size**.

:hash-function, :compare-function, funcallablep — These arguments are for internal use only.

map *fn* {*list*}* Function

This function is incompatible with the Common Lisp **map** function described in Section 9, Sequences. This Zetalisp **map** is synonymous with the **mapl** function described in Section 14, Control Structures.

member *item list* Function

The Zetalisp version of the **member** function uses an **equal** predicate rather than **eql**, which is the Common Lisp default. Note the following equivalence:

```
(zlc::member elm trees) <=> (lisp:member elm trees :test #'equal)
```

named-structure-invoke *operation instance* &rest *args* Function

This function invokes a named structure operation on *instance*. The *operation* argument should be a keyword symbol, and *instance* should be a named structure. The handler function of the named structure symbol, found as the value of the **named-structure-invoke** property of the symbol, is called with appropriate arguments. (This function used to take its first two arguments in the opposite order, and that argument order will continue to work indefinitely, but it should not be used in new programs.)

If the structure type has no **named-structure-invoke** property, **nil** is returned.

The (**send** *instance operation args...*) form has the same effect by calling **named-structure-invoke**.

See also the **:named-structure-symbol** keyword to **make-array**.

nlistp *object* Function

This Zetalisp predicate is the same as the Common Lisp function **atom** described in Section 6, Lists and List Structures. In Common Lisp mode, **nlistp** differs by returning **nil** when the argument is **nil**.

rassoc *item a-list* Function

This function is like **rassoc** in Section 6, Lists and List Structures, except that the test function is always **equal**.

read &optional *input-stream eof-option rubout-handle-options* Function

See the Common Lisp **read** function in the *Explorer Input/Output Reference* manual.

read-from-string *string* &optional *eof-option start end* Function

This function is like the Common Lisp function of the same name in the *Explorer Input/Output Reference* manual except that it does not provide the *eof-error-p* and **:preserve-whitespace** options.

rem *predicate item list* &optional *n* Function

This function is the same as **remq** but differs in that it takes an extra argument that should be a predicate of two arguments, which is used for the comparison test instead of **eq**.

remove *item list* &optional *n* Function

The Zetalisp **remove** function is like the Common Lisp **remove**, but it operates on lists only and does not make use of the keywords. The &optional argument *n* operates exactly like the keyword **:count**, and **equal** is always used as the test predicate.

some *list predicate* &optional *step-function* Function

This function returns a portion of *list* beginning with the first element for which *predicate* returns a non-**nil** value. If *predicate* returns **nil** for every element of *list*, **some** returns **nil**. If *step-function* is supplied, it is used to get the next element of *list* (rather than **cdr**).

string *x* Function

This function is like the Common Lisp **string** function (see Section 8, Strings) except that the *x* argument can also be an integer, which is coerced to a **string-char** character.

subst *new old tree* Function

This function is like **subst** in Section 6, Lists and List Structure, except that the test function is always **equal** and the **:key** optional keyword is unavailable.

union *list1* &rest *more-lists* Function
nunion *list1* &rest *more-lists* Function

These Zetalisp functions are like the Common Lisp functions of the same name in Section 6 except that any number of lists can be supplied as arguments and the test predicate is always **eq**.

Other Considerations

A.2 The USER package uses the ZLC, TICL, and LISP packages. The default for **make-package**, however, does not use the ZLC package. Other differences between Zetalisp mode and Common Lisp modes are as follows:

- Each mode has its own readtable. Reader macros may behave differently in the two modes. For instance, **#/***char* reads a character in Zetalisp mode, but you must use **#***char* to read a character in Common Lisp mode.
- Zetalisp uses / as the escape character, whereas Common Lisp uses \\. This means, for example, that the division function is written as / in Common Lisp mode, but it is // in Zetalisp mode.
- In Zetalisp mode, characters are represented as fixnums, but in Common Lisp mode they are character objects.
- The editing mode for the default Zmacs buffer is the current mode, whether Zetalisp or Common Lisp.

INDEX

Introduction

The indexes for the *Explorer Lisp Reference* and the *Explorer Input/Output Reference* have been combined for ease of use. Index entries are denoted with IO or LISP preceding the page number.

The indexes for this Explorer software manual are divided into several subindexes. Each subindex contains all the entries for a particular category, such as functions, variables, or concepts. The various subindexes for this manual and the pages on which they begin are as follows:

Alphabetization Scheme

The alphabetization scheme used in this index ignores package names and nonalphabetic symbol prefixes for the purposes of sorting. For example, the **rpc:*callrpc-retrys*** variable is sorted under the entries for the letter C rather than under the letter R.

Hyphens are sorted after spaces. Consequently, the multiple menus entry precedes the multiple-choice facility entry. However, the apropos-flavor entry precedes the aproposb entry, as follows:

apropos, 25-7
apropos-flavor, 25-9
aproposb, 25-9

Underscore characters are sorted after hyphens. Consequently, the **xdr-io** macro precedes the **xdr_destroy** macro.

General

Conditions

Flavors

Functions

Operations

Variables

Data Systems Group – Austin
Documentation Questionnaire

Explorer Lisp Reference

Do you use other TI manuals? If so, which one(s)?

______________________ ______________________

______________________ ______________________

______________________ ______________________

How would you rate the quality of our manuals?

	Excellent	Good	Fair	Poor
Accuracy	____	____	____	____
Organization	____	____	____	____
Clarity	____	____	____	____
Completeness	____	____	____	____
Overall design	____	____	____	____
Size	____	____	____	____
Illustrations	____	____	____	____
Examples	____	____	____	____
Index	____	____	____	____
Binding method	____	____	____	____

Was the quality of documentation a criterion in your selection of hardware or software?

☐ Yes ☐ No

How do you find the technical level of our manuals?

☐ Written for a more experienced user than yourself
☐ Written for a user with the same experience
☐ Written for a less experienced user than yourself

What is your experience using computers?

☐ Less than 1 year ☐ 1-5 years ☐ 5-10 years ☐ Over 10 years

We appreciate your taking the time to complete this questionnaire. If you have additional comments about the quality of our manuals, please write them in the space below. Please be specific.

____

____

____

____

____

Name ______________________ Title/Occupation ______________________

Company Name ____

Address ______________________ City/State/Zip ______________________

Telephone ______________________ Date ______________________

TAPE EDGE TO SEAL

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DATA SYSTEMS GROUP
ATTN: PUBLISHING CENTER
P.O. Box 2909 M/S 2146
Austin, Texas 78769-9990

FOLD